C++ Programming

Chapter 1 - C++ Overview

What is C++?

- C++ was developed by Bjarne Stroustrup, as an extension to the C language.
- Despite being an 80s creation, C++ has been a popular programming language throughout these years.
- C++ is a cross-platform language that can be used to create high-performance applications and software systems.
- C++ is very close to the hardware making it comparatively easy for programmers to give the instructions directly to the system without any intermediary giving programmers a high level of control over system resources and memory.

Why should we learn C++/ Features of C++?

- C++ is one of the world's most popular programming languages.
- In today's operating systems, GUIs, and embedded systems, C++ is widely used.
- It is one of the most popular programming languages for its object-orientedness. C++ is an object-oriented programming language that gives a clear structure to programs and allows code to be reused, lowering development costs.
- With C++, you can develop applications or heavy games that can run on different platforms.
- As C++ is close to other programming languages such as C# and Java, which makes it easy for programmers to switch to C++ or vice versa while it is actually very easy to learn.

How is it different from C?

- The syntax of C++ is almost identical to that of C, as C++ was developed as an extension of C.
- In contrast to C, C++ supports classes and objects, while C does not.

Chapter 2 - Getting Started with C++

Requirements before you start

To start using C++, you need two things:

- A text editor, like Notepad, or an IDE, like VSCode to act as a platform for you to write C++ code
- A compiler, like GCC to translate the C++ code you have written which is a high-level language into a low-level language that the computer will understand.

What is an IDE?

- IDE stands for Integrated Development Environment.
- It is nothing more than an enhanced version of a text editor that helps you write more efficient and nicer code.
- It helps to differentiate different parts of your codes with different colors and notifies you if you are missing some semicolon or bracket at some place by highlighting that area.
- A lot of IDEs are available, such as DEVC++ or Code Blocks, but we will prefer using VS Code for this tutorial series.

Installing VSCode

- Visit https://code.visualstudio.com/download
- Click on the download option as per your operating system.
- After the download is completed, open the setup and run it by saving VS Code in the default location without changing any settings.
- You will need to click the next button again and again until the installation process begins.

What is a Compiler?

- A compiler is used to run the program of a certain language which is generally high-level by converting the code into a language that is low-level that our computer could understand.
- There are a lot of compilers available, but we will proceed with teaching you to use MinGW for this course because it will fulfill all of our requirements, and also it is recommended by Microsoft itself.

Setting up the compiler

- Visit https://code.visualstudio.com/docs/languages/cpp
- Select C++ from the sidebar.
- Choose “GCC via Mingw-w64 on Windows” from the options shown there.
- Select the install sourceforge option.
- After the downloading gets completed, run the setup and choose all the default options as we did while installing VS Code.

Setting Path for Compiler

- Go to the C directory. Navigate into the Program Files. Then, open MinGW-64. Open MinGW-32. And then the bin folder. After reaching the bin, save the path or URL to the bin.

- Then go to the properties of 'This PC'.
- Select 'Advance System Settings'.
- Select the 'Environment Variable' option.
- Add the copied path to the Environment Variable.
- And now, you can visit your IDE and run your C++ programs on it. The configuration part is done.

Writing your first code in C++

Open VSCode. Here's the simplest print statement we can start with.

```
#include <iostream>

int main()
{
    std::cout << "Hello World";
    return 0;
}
```

Output:

```
Hello World
```

Chapter 3 - Basic Structure & Syntax

Programming in C++ involves following a basic structure throughout. To understand that basic structure, the first program we learned writing in C++ will be referred to.

```
#include <iostream>

int main()
{
    std::cout << "Hello World";
    return 0;
}
```

Here's what it can be broken down to.

Pre-processor commands/ Header files

- It is common for C++ programs to include many built-in elements from the standard C++ library, including classes, keywords, constants, operators, etc. It is necessary to include an appropriate header file in a program in order to use such pre-defined elements.
- In the above program, #include was the line put to include the header file iostream. The iostream library helps us to get input data and show output data. The iostream library also has many more uses and error facilities; it is not only limited to input and output.
- Header file are both system defined and user defined. To know more about header files, go to the documentary here, https://en.cppreference.com/w/cpp/header.

Definition Section

Here, all the variables, or other user-defined data types are declared. These variables are used throughout the program and all the functions.

Function Declaration

- After the definition of all the other entities, here we declare all the functions a program needs. These are generally user-defined.
- Every program contains one main parent function which tells the compiler where to start the execution of the program.
- All the statements that are to be executed are written in the main function.
- Only the instructions enclosed in curly braces {} are considered for execution by the compiler.
- After all instructions in the main function have been executed, control leaves the main function and the program ends.

A C++ program is made up of different tokens combined. These tokens include:

- Keywords
- Identifiers
- Constants
- String Literal
- Symbols & Operators

1. Keywords

- Keywords are reserved words that can not be used elsewhere in the program for naming a variable or a function. They have a specific function or task and they are solely used for that. Their functionalities are pre-defined.
- One such example of a keyword could be return which is used to build return statements for functions. Other examples are auto, if, default, etc.
- There is a list of reserved keywords which cannot be reused by the programmer or overloaded. One can find the list here, https://en.cppreference.com/w/cpp/keyword

2. Identifiers

- Identifiers are names given to variables or functions to differentiate them from one another. Their definitions are solely based on our choice but there are a few rules that we have to follow while naming identifiers. One such rule says that the name can not contain special symbols such as @, -, *, <, etc.
- C++ is a case-sensitive language so an identifier containing a capital letter and another one containing a small letter in the same place will be different. For example, the three words: Code, code, and cOde can be used as three different identifiers.

3. Constants

- Constants are very similar to a variable and they can also be of any data type. The only difference between a constant and a variable is that a constant's value never changes. We will see constants in more detail in the upcoming chapters.

4. String Literal

- String literals or string constants are a sequence of characters enclosed in double quotation marks. Escape sequences are also string literals.

5. Symbols and Operators

- Symbols are special characters reserved to perform certain actions. Using them lets the compiler know what specific tasks should be performed on the given data. Several examples of symbols are arithmetical operators such as +, *, or bitwise operators such as ^, &.

Chapter 4 - C++ Comments

A comment is a human-readable text in the source code, which is ignored by the compiler. Comments can be used to insert any informative piece which a programmer does not wish to be executed. It could be either to explain a piece of code or to make it more readable. In addition, it can be used to prevent the execution of alternative code when the process of debugging is done.

Comments can be singled-lined or multi-lined.

Single Line Comments

- Single-line comments start with two forward slashes (//).
- Any information after the slashes // lying on the same line would be ignored (will not be executed) since they become unparsable.

An example of how we use a single-line comment

```
#include <iostream>

int main()
{
    // This is a single line comment
    std::cout << "Hello World";
    return 0;
}
```

Multi-line comments

- A multi-line comment starts with /_ and ends with _/.
- Any information between /_ and _/ will be ignored by the compiler.

An example of how we use a multi-line comment

```
#include <iostream>

int main()
{
    /* This is a
    multi-line
    comment */

    std::cout << "Hello World";
    return 0;
}
```

Chapter 5 - C++ Variables

Variables are containers for storing data values.

In C++, there are different types of variables.

Some of them are as follows:

- an integer variable defined with the keyword int stores integers (whole numbers), without decimals, such as 63 or -1.
- a floating point variable defined with keyword float stores floating point numbers, with decimals, such as 79.97 or -13.26.
- a character variable defined with the keyword char stores single characters, such as 'A' or 'z'. Char values are bound to be surrounded by single quotes.
- a boolean variable defined with the keyword bool stores a single value 0 or 1 for false and true respectively.

Declaration

We cannot declare a variable without specifying its data type. The data type of a variable depends on what we want to store in the variable and how much space we want it to hold.

The syntax for declaring a variable is simple:

```
data_type  variable_name;
```

OR

```
data_type  variable_name = value;
```

Naming a Variable

There is no limit to what we can call a variable. Yet there are specific rules we must follow while naming a variable:

- A variable name in C++ can have a length of range 1 to 255 characters
- A variable name can only contain alphabets, digits, and underscores(_).
- A variable cannot start with a digit.
- A variable cannot include any white space in its name.
- Variable names are case sensitive
- The name should not be a reserved keyword or any special character.

Variable Scope

- The scope of a variable is the region in a program where the existence of that variable is valid. Based on its scope, variables can be classified into two types:

  Local variables:

  - Local variables are declared inside the braces of any function and can be assessed only from that particular function.

  Global variables:

  - Global variables are declared outside of any function and can be accessed from anywhere.

An example that demonstrates the difference in applications of a local and a global variable is given below.

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

int a = 5; //global variable

void func()
{
    cout << a << endl;
}

int main()
{
    int a = 10; //local variable
    cout << a << endl;
    func();
    return 0;
}
```

Output:

```
10
5
```

***Explanation**: A local variable a was declared in the main function, and when printed, gave 10. This is because, within the body of a function, a local variable takes precedence over a global variable with the same name. But since there was no variable declared in the func function, it considered the global variable a for printing, and hence the value 5.*

A variable, as its name is defined, can be altered, or its value can be changed, but the same is not true for its type. If a variable is of integer type, it will only store an integer value through a program. We cannot assign a character type value to an integer variable. We can not even store a decimal value into an integer variable.

Chapter 6 - C++ Data Types & Constants

C++ Data Types

Data types define the type of data a variable can hold; for example, an integer variable can hold integer data, a character can hold character data, etc.

Data types in C++ are categorized into three groups:

Built-in data types

These data types are pre-defined for a language and could be used directly by the programmer.

Examples are: Int, Float, Char, Double, Boolean

User-defined data types

These data types are defined by the user itself.

Examples are: Class, Struct, Union, Enum

Derived data types

These data types are derived from the primitive built-in data types.

Examples are: Array, Pointer, Function

Some of the popular built-in data types and their applications are:

Data Type	**Size**	**Description**
int	2 or 4 bytes	Stores whole numbers, without decimals
float	4 bytes	Stores fractional numbers, containing one or more decimals. They require 4 bytes of memory space.
double	8 bytes	Stores fractional numbers, containing one or more decimals. They require 4 bytes of memory space.
char	1 byte	Stores a single character/letter/number, or ASCII values
boolean	1 byte	Stores true or false values

C++ Constants

Constants are unchangeable; when a constant variable is initialized in a program, its value cannot be changed afterwards.

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

int main()
{
    const float PI = 3.14;
    cout << "The value of PI is " << PI << endl;
    PI = 3.00; //error, since changing a const variable is not allowed.
}
```

Output:

```
error: assignment of read-only variable 'PI'
```

Chapter 7 - C++ Operators

Special symbols that are used to perform actions or operations are known as operators. They could be both unary or binary.

For example, the symbol + is used to perform addition in C++ when put in between two numbers, so it is a binary operator. There are different types of operators. They are as follows:

Arithmetic Operators

Arithmetic operators are used to perform mathematical operations such as addition, subtraction, etc. They could be both binary and unary. A few of the simple arithmetic operators are

Operation	Description
a + b	Adds a and b
a - b	Subtracts b from a
a * b	Multiplies a and b
a / b	Divides a by b
a % b	Modulus of a and b
a++	Post increments a by 1
a--	Post decrements a by 1
++a	Pre increments a by 1
--a	Pre decrements a by 1

Let's see their implementation in C++.

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

int main()
{
    int a = 4, b = 5;
    cout << "The value of a + b is " << a + b << endl;
    cout << "The value of a - b is " << a - b << endl;
    cout << "The value of a * b is " << a * b << endl;
    cout << "The value of a / b is " << a / b << endl;
    cout << "The value of a % b is " << a % b << endl;
    cout << "The value of a++ is " << a++ << endl;
    cout << "The value of a-- is " << a-- << endl;
    cout << "The value of ++a is " << ++a << endl;
    cout << "The value of --a is " << --a << endl;
}
```

Output:

```
The value of a + b is 9
The value of a - b is -1
The value of a * b is 20
The value of a / b is 0
The value of a % b is 4
The value of a++ is 4
The value of a-- is 5
The value of ++a is 5
The value of --a is 4
```

Relational Operators

Relational operators are used to check the relationship between two operands and to compare two or more numbers or even expressions in cases. The return type of a relational operator is a Boolean that is, either True or False (1 or 0).

Operator	Description
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
==	Is equal to
!=	Is not equal to

Let's see their implementation in C++.

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

int main()
{
    int a = 4, b = 5;
    cout << "The value of a == b is " << (a == b) << endl;
    cout << "The value of a < b is " << (a < b) << endl;
    cout << "The value of a > b is " << (a > b) << endl;
}
```

Output:

```
The value of a==b is 0
The value of a<b is 1
```

```
The value of a>b is 0
```

The output is 0 for a==b, since a and b are not equal and 1 for a<b, since a is less than b.

Logical Operators

Logical Operators are used to check whether an expression is true or false. There are three logical operators i.e. AND, OR, and NOT. They can be used to compare Boolean values but are mostly used to compare expressions to see whether they are satisfying or not.

- AND: it returns true when both operands are true or 1.
- OR: it returns true when either operand is true or 1.
- NOT: it is used to reverse the logical state of the operand and is true when the operand is false.

Operator	Description
&&	AND Operator
||	OR Operator
!	NOT Operator

Let's see their implementation in C++.

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

int main()
{
    int a = 1, b = 0;
    cout << "The value of a && b is " << (a && b) << endl;
    cout << "The value of a || b is " << (a || b) << endl;
    cout << "The value of !a is " << (!a) << endl;
}
```

Output:

```
The value of a && b is 0
The value of a || b is 1
The value of !a is 0
```

Bitwise Operators

A bitwise operator is used to perform operations at the bit level. To obtain the results, they convert our input values into binary format and then process them using whatever operator they are being used with.

Operator	Description
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR
~	Bitwise Complement
>>	Shift Right Operator
<<	Shift Left Operator

Let's see their implementation in C++.

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

int main()
{
    int a = 13; //1101
    int b = 5;  //101
    cout << "The value of a & b is " << (a & b) << endl;
    cout << "The value of a | b is " << (a | b) << endl;
    cout << "The value of a ^ b is " << (a ^ b) << endl;
    cout << "The value of ~a is " << (~a) << endl;
    cout << "The value of a >> 2 is " << (a >> 2) << endl;
    cout << "The value of a << 2 is " << (a << 2) << endl;
}
```

Output:

```
The value of a & b is 5
The value of a | b is 13
The value of a ^ b is 8
The value of ~a is -14
The value of a >> 2 is 3
The value of a << 2 is 52
```

Assignment Operators

Assignment operators are used to assign values. We will use them in almost every program we develop.

```
int a = 0;
int b = 1;
```

Equal to (=) is the assignment operator here. It is assigning 0 to a and 1 to b in the above example.

Operator	Description
=	It assigns the right side operand value to the left side operand.
+=	It adds the right operand to the left operand and assigns the result to the left operand.
-=	It subtracts the right operand from the left operand and assigns the result to the left operand.
*=	It multiplies the right operand with the left operand and assigns the result to the left operand.
/=	It divides the left operand with the right operand and assigns the result to the left operand.

Operator Precedence and Associativity

Operator precedence

- It helps us determine the precedence of an operator over another while solving an expression. Consider an expression a+b*c. Now, since the multiplication operator's precedence is higher than the precedence of the addition operator, multiplication between a and b is done first and then the addition operation will be performed.

Operator associativity

- It helps us to solve an expression; when two or more operators having the same precedence come together in an expression. It helps us decide whether we should start solving the expression containing operators of the same precedence from left to right or from right to left.

The table containing the operator precedence and operator associativity of all operators can be found here. (C++ Operator Precedence)[https://en.cppreference.com/w/cpp/language/operator_precedence].

Chapter 8 - C++ Manipulators

In C++ programming, language manipulators are used in the formatting of output. These are helpful in modifying the input and the output stream. They make use of the insertion and extraction operators to modify the output.

Here's a list of a few manipulators:

Manipulators	Description
endl	It is used to enter a new line with a flush.
setw(a)	It is used to specify the width of the output.
setprecision(a)	It is used to set the precision of floating-point values.
setbase(a)	It is used to set the base value of a numerical number.

Let's see their implementation in C++. Note that we use the header file iomanip for some of the manipulators.

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

int main()
{
    float PI = 3.14;
    int num = 100;
    cout << "Entering a new line." << endl;
    cout << setw(10) << "Output" << endl;
    cout << setprecision(10) << PI << endl;
    cout << setbase(16) << num << endl; //sets base to 16
}
```

Output:

```
Entering a new line.
    Output
3.140000105
64
```

Chapter 9 - C++ Basic Input/Output

C++ language comes with different libraries, which help us in performing input/output operations. In C++, sequences of bytes corresponding to input and output are commonly known as streams. There are two types of streams.

They are,

Input stream

In the input stream, the direction of the flow of bytes occurs from the input device (for ex keyboard) to the main memory.

Output stream

In the output stream, the direction of flow of bytes occurs from the main memory to the output device (for ex-display)

An example that demonstrates how input and output are popularly done in C++.

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

int main()
{
    int num;
    cout << "Enter a number: ";
    cin >> num;                          // Getting input from the user
    cout << "Your number is: " << num; // Displaying the input value
    return 0;
}
```

Input:

```
Enter a number: 10
```

Output:

```
Your number is: 10
```

Important Points

- The sign << is called the insertion operator.
- The sign >> is called the extraction operator.

- `cout` keyword is used to print.
- `cin` keyword is used to take input at run time.

Chapter 10 - Control Structure

The work of control structures is to give flow and logic to a program. There are three types of basic control structures in C++.

Sequence Structure

Sequence structure refers to the sequence in which program execute instructions one after another.

Selection Structure

Selection structure refers to the execution of instruction according to the selected condition, which can be either true or false. There are two ways to implement selection structures. They are done either by if-else statements or by switch case statements.

Loop Structure

Loop structure refers to the execution of an instruction in a loop until the condition gets false.

Chapter 11 - C++ If Else

If else statements are used to implement a selection structure. Like any other programming language, C++ also uses the if keyword to implement the decision control instruction.

The condition for the if statement is always enclosed within a pair of parentheses. If the condition is true, then the set of statements following the if statement will execute. And if the condition evaluates to false, then the statement will not execute, instead, the program skips that enclosed part of the code.

An expression in if statements are defined using relational operators. The statement written in an if block will execute when the expression following if evaluates to true. But when the if block is followed by an else block, then when the condition written in the if block turns to be false, the set of statements in the else block will execute.

Following is the syntax of if-else statements:

```
if ( condition ){
statements;}
 else {
statements;}
```

One example where we could use the if-else statement is:

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

int main()
{
    int age;
    cout << "Enter a number: ";
    cin >> age;
    if (age >= 50)
    {
        cout << "Input number is greater than 50!" << endl;
    }
    else if (age == 50)
    {
        cout << "Input number is equal to 50!" << endl;
    }
    else
    {
        cout << "Input number is less than 50!" << endl;
    }
}
```

Input:

```
Enter a number: 51
```

Output:

```
Input number is greater than 50!
```

> **Note:** The else if statement checks for a different condition if the conditions checked above it evaluate to false.

Chapter 12 - C++ Switch Case

The control statement that allows us to make a decision effectively from the number of choices is called a switch, or a switch case-default since these three keywords go together to make up the control statement.

Switch executes that block of code, which matches the case value. If the value does not match with any of the cases, then the default block is executed.

Following is the syntax of switch case-default statements:

```
switch ( integer/character expression )
{
case {value 1} :
do this ;

case {value 2} :
 do this ;

default :
do this ;
 }
```

The expression following the switch can be an integer expression or a character expression. Remember, that case labels should be unique for each of the cases. If it is the same, it may create a problem while executing a program. At the end of the case labels, we always use a colon (:). Each case is associated with a block. A block contains multiple statements that are grouped together for a particular case.

The break keyword in a case block indicates the end of a particular case. If we do not put the break in each case, then even though the specific case is executed, the switch will continue to execute all the cases until the end is reached. The default case is optional. Whenever the expression's value is not matched with any of the cases inside the switch, then the default case will be executed.

One example where we could use the switch case statement is

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

int main()
{
    int i = 2;
    switch (i)
    {
    case 1:
        cout << "Statement 1" << endl;
        break;

    case 2:
        cout << "Statement 2" << endl;
```

```
            break;

        default:
            cout << "Default statement!" << endl;
        }
    }
```

Output:

```
Statement 2
```

The test expression of a switch statement must necessarily be of an integer or character type and the value of the case should be an integer or character as well. Cases should only be inside the switch statement and using the break keyword in the switch statement is not necessary.

Chapter 13 - C++ Loops

The need to perform an action, again and again, with little or no variations in the details each time they are executed is met by a mechanism known as a loop. This involves repeating some code in the program, either a specified number of times or until a particular condition is satisfied. Loop-controlled instructions are used to perform this repetitive operation efficiently ensuring the program doesn't look redundant at the same time due to the repetitions.

Following are the three types of loops in C++ programming.

- For Loop
- While Loop
- Do While Loop

Chapter 14 - For Loop

A for loop is a repetition control structure that allows us to efficiently write a loop that will execute a specific number of times. The for-loop statement is very specialized. We use a for loop when we already know the number of iterations of that particular piece of code we wish to execute. Although, when we do not know about the number of iterations, we use a while loop which is discussed next.

Here is the syntax of a for loop in C++ programming.

```
for (initialise counter; test counter; increment / decrement counter)
{
    //set of statements
}
```

Here,

- initialize counter: It will initialize the loop counter value. It is usually i=0.
- test counter: This is the test condition, which if found true, the loop continues, otherwise terminates.
- Increment/decrement counter: Incrementing or decrementing the counter.
- Set of statements: This is the body or the executable part of the for loop or the set of statements that has to repeat itself.

One such example to demonstrate how a for loop works is

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

int main()
{
    int num = 10;
    int i;
    for (i = 0; i < num; i++)
    {
        cout << i << " ";
    }
    return 0;
}
```

Output:

```
0 1 2 3 4 5 6 7 8 9
```

First, the initialization expression will initialize loop variables. The expression i=0 executes once when the loop starts. Then the condition i < num is checked. If the condition is true, then the statements inside the body of the loop are executed. After the statements inside the body are executed, the control of the program is transferred to the increment of the variable i by 1. The expression i++ modifies the loop variables. Iteratively, the condition i < num is evaluated again.

The for loop terminates when i finally becomes greater than num, therefore, making the condition `i<num false`.

Chapter 15 - While Loop

A While loop is also called a pre-tested loop. A while loop allows a piece of code in a program to be executed multiple times, depending upon a given test condition which evaluates to either true or false. The while loop is mostly used in cases where the number of iterations is not known. If the number of iterations is known, then we could also use a for loop as mentioned previously.

Following is the syntax for using a while loop.

```
while (condition test)
{
    // Set of statements
}
```

The body of a while loop can contain a single statement or a block of statements. The test condition may be any expression that should evaluate as either true or false. The loop iterates while the test condition evaluates to true. When the condition becomes false, it terminates.

One such example to demonstrate how a while loop works is

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

int main()
{
    int i = 5;
    while (i < 10)
    {
        cout << i << " ";
        i++;
    }

    return 0;
}
```

Output:

```
5 6 7 8 9
```

Chapter 16 - Do While Loop

A do-while loop is a little different from a normal while loop. A do-while loop, unlike what happens in a while loop, executes the statements inside the body of the loop before checking the test condition.

So even if a condition is false in the first place, the do-while loop would have already run once. A do-while loop is very much similar to a while loop, except for the fact that it is guaranteed to execute the body at least once.

Unlike for and while loops, which test the loop condition first, then execute the code written inside the body of the loop, the do-while loop checks its condition at the end of the loop.

Following is the syntax for using a do-while loop.

```
do
{
    statements;
} while (test condition);
```

First, the body of the do-while loop is executed once. Only then, the test condition is evaluated. If the test condition returns true, the set of instructions inside the body of the loop is executed again, and the test condition is evaluated. The same process goes on until the test condition becomes false. If the test condition returns false, then the loop terminates.

One such example to demonstrate how a do-while loop works is

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

int main()
{
    int i = 5;
    do
    {
        cout << i << " ";
        i++;
    } while (i < 5);

    return 0;
}
```

Output:

```
5
```

Here, even if i was less than 5 from the very beginning, the do-while let the print statement execute once, and then terminated.

Chapter 17 - Break Statement

Break statement is used to break the loop or switch case statements execution and brings the control to the next block of code after that particular loop or switch case it was used in.

Break statements are used to bring the program control out of the loop it was encountered in. The break statement is used inside loops or switch statements in C++ language.

One such example to demonstrate how a break statement works is

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

int main()
{
    int num = 10;
    int i;
    for (i = 0; i < num; i++)
    {
        if (i == 6)
        {
            break;
        }
        cout << i << " ";
    }

    return 0;
}
```

Output:

```
0 1 2 3 4 5
```

Here, when i became 6, the break statement got executed and the program came out of the for loop.

Chapter 18 - Continue Statement

The continue statement is used inside loops in C++ language. When a continue statement is encountered inside the loop, the control jumps to the beginning of the loop for the next iteration, skipping the execution of statements inside the body of the loop after the continue statement.

It is used to bring the control to the next iteration of the loop. Typically, the continue statement skips some code inside the loop and lets the program move on with the next iteration. It is mainly used for a condition so that we can skip some lines of code for a particular condition.

It forces the next iteration to follow in the loop unlike a break statement, which terminates the loop itself the moment it is encountered.

One such example to demonstrate how a continue statement works is

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

int main()
{
    for (int i = 0; i <= 10; i++)
    {
        if (i < 6)
        {
            continue;
        }
        cout << i << " ";
    }
    return 0;
}
```

Output:

```
6 7 8 9 10
```

Here, the continue statement was continuously executing while i remained less than 5. For all the other values of i, we got the print statement working.

Chapter 19 - Array Basics

An array is a collection of items that are of the data type stored in contiguous memory locations. And it is also known as a subscript variable.

It can even store the collection of derived data types such as pointers, structures, etc.

An array can be of any dimension. The C++ Language places no limits on the number of dimensions in an array. This means we can create arrays of any number of dimensions. It could be a 2D array or a 3D array or more.

Advantages of Arrays?

- It is used to represent multiple data items of the same type by using only a single name.
- Accessing any random item at any random position in a given array is very fast in an array.
- There is no case of memory shortage or overflow in the case of arrays since the size is fixed and elements are stored in contiguous memory locations.

Chapter 20 - Array Operations

Defining an array

1. Without specifying the size of the array:

```
int arr[] = {1, 2, 3};
```

Here, we can leave the square brackets empty, although the array cannot be left empty in this case. It must have elements in it.

2. With specifying the size of the array:

```
int arr[3];
arr[0] = 1, arr[1] = 2, arr[2] = 3;
```

Accessing an array element

An element in an array can easily be accessed through its index number.

An index number is a special type of number which allows us to access variables of arrays. Index number provides a method to access each element of an array in a program. This must be remembered that the index number starts from 0 and not one.

Example:

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

int main()
{
    int arr[] = {1, 2, 3};
    cout << arr[1] << endl;
}
```

Output:

```
2
```

Changing an array element

An element in an array can be overwritten using its index number.

Example:

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

int main()
{
    int arr[] = {1, 2, 3};
    arr[2] = 8; //changing the element on index 2
    cout << arr[2] << endl;
}
```

Output:

```
8
```

Chapter 21 - Pointers

A pointer is a data type that holds the address of another data type. A pointer itself is a variable that points to any other variable. It can be of type int, char, array, function, or even any other pointer. Pointers in C++ are defined using the '*' (asterisk) operator.

The '&'(ampersand) operator is called the 'address of' operator, and the '*'(asterisk) operator is called the 'value at' dereference operator.

Applications of a Pointer

- Pointers are used to dynamically allocate or deallocate memory.
- Pointers are used to point to several containers such as arrays, or structs, and also for passing addresses of containers to functions.
- Return multiple values from a function
- Rather than passing a copy of a container to a function, we can simply pass its pointer. This helps reduce the memory usage of the program.
- Pointer reduces the code and improves the performance.

Chapter 22 - Operations on Pointers

Address of Operator (&):

& is also known as the Referencing Operator. It is a unary operator. The variable name used along with the Address of operator must be the name of an already defined variable.

Using & operator along with a variable gives the address number of the variable.

Here's one example to demonstrate the use of the address of the operator.

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

int main()
{
    int a = 10;
    cout << "Address of variable a is " << &a << endl;
    return 0;
}
```

Output:

```
0x61febc
```

Indirection Operator

- * is also known as the Dereferencing Operator. It is a unary operator. It takes an address as its argument and returns the content/container whose address is its argument.

Here's one example to demonstrate the use of the indirection operator.

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

int main()
{
    int a = 100;
    cout << "Value of variable a stored at address " << &a << " is " << (*(&a)) <<
endl;
    return 0;
}
```

Output:

```
Value of variable a stored at address 0x61febc is 100
```

Pointer to Pointer

Pointer to Pointer is a simple concept, in which we store the address of one pointer to another pointer. This is also known as multiple indirections owing to the operator's name. Here, the first pointer contains the address of the second pointer, which points to the address where the actual variable has its value stored.

An example to demonstrate how we define a pointer to a pointer.

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

int main()
{
    int a = 100;
    int *b = &a;
    int **c = &b;
    cout << "Value of variable a is " << a << endl;
    cout << "Address of variable a is " << b << endl;
    cout << "Address of pointer b is " << c << endl;
    return 0;
}
```

Output:

```
Value of variable a is 100
Address of variable a is 0x61feb8
Address of pointer b is 0x61feb4
```

Arrays and Pointers

Storing the address of an array into pointer is different from storing the address of a variable into the pointer. The name of an array itself is the address of the first index of an array. So, to use the (ampersand)& operator with the array name for assigning the address to a pointer is wrong. Instead, we used the array name itself.

An example program for storing the starting address of an array in the pointer,

```
int marks[] = {99, 100, 38};
int *p = marks;
cout << "The value of marks[0] is " << *p << endl;
```

Output:

```
The value of marks[0] is 99
```

In order to access other elements of the same array that pointer p points to, we can use pointer arithmetic, such as addition and subtraction of pointers.

`*(p+1)` returns the value at the second position in the array marks. Here's how it works.

```
int marks[] = {99, 100, 38};
int *p = marks;
cout << "The value of marks[0] is " << *p << endl;
cout << "The value of marks[1] is " << *(p + 1) << endl;
cout << "The value of marks[2] is " << *(p + 2) << endl;
```

Output:

```
The value of marks[0] is 99
The value of marks[1] is 100
The value of marks[2] is 38
```

Chapter 23 - Strings

A string is an array of characters. Unlike in C, we can define a string variable and not necessarily a character array to store a sequence of characters. Data of the same type are stored in an array, for example, integers can be stored in an integer array, similarly, a group of characters can be stored in a character array or a string variable. A string is a one-dimensional array of characters.

Declaring a string is very simple, the same as declaring a one-dimensional array. It's just that we are considering it as an array of characters.

Below is the syntax for declaring a string.

```
string string_name ;
```

In the above syntax, string_name is any name given to the string variable and it can be given a string input later or it can even be initialized at the time of definition.

```
string string_name = "Hello World";
```

Example of a string:

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

int main()
{
    // declare and initialise string
    string str = "Hello World";
    cout << str << endl;
    return 0;
}
```

Output:

```
Hello World
```

> **Note:** To be able to use these strings, you must declare another header file named string. It contains a lot of useful string functions and libraries as well.

Chapter 24 - Structures

Variables store only one piece of information and arrays of certain data types store the information of the same data type. Variables and arrays can handle a great variety of situations. But quite often, we have to deal with the collection of dissimilar data types. For dealing with cases where there is a requirement to store dissimilar data types, C++ provides a data type called 'structure'. The structure is a user-defined data type that is available in C++. Structures are used to combine different types of data types, just like an array is used to combine the same type of data types.

Another way to store data of dissimilar data types would have been constructing individual arrays, but that must be unorganized. It is to keep in mind that structure elements too are always stored in contiguous memory locations.

Features of Structs

1. We can assign the values of a structure variable to another structure variable of the same type using the assignment operator.
2. Structure can be nested within another structure which means structures can have their members as structures themselves.
3. We can pass the structure variable to a function. We can pass the individual structure elements or the entire structure variable into the function as an argument. And functions can also return a structure variable.
4. We can have a pointer pointing to a struct just like the way we can have a pointer pointing to an int, or a pointer pointing to a char variable.

Creating a struct element

We use the struct keyword to define the struct.

The basic syntax for declaring a struct is,

```
struct structure_name
{
    //structure_elements
} structure_variable;
```

Here's one example of how a struct is defined and used in main as a user-defined data type.

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

struct employee
{
    /* data */
```

```
    int eId;
    char favChar;
    int salary;
};

int main()
{
    struct employee Harry;
    return 0;
}
```

Accessing struct elements

To access any of the values of a structure's members, we use the dot operator (.). This dot operator is coded between the structure variable name and the structure member that we wish to access.

Before the dot operator, there must always be an already defined structure variable and after the dot operator, there must always be a valid structure element.

Here's one example demonstrating how we access struct elements.

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

struct employee
{
    /* data */
    int eId;
    char favChar;
    int salary;
};

int main()
{
    struct employee Harry;
    Harry.eId = 1;
    Harry.favChar = 'c';
    Harry.salary = 120000000;
    cout << "eID of Harry is " << Harry.eId << endl;
    cout << "favChar of Harry is " << Harry.favChar << endl;
    cout << "salary of Harry is " << Harry.salary << endl;
    return 0;
}
```

Output:

```
eID of Harry is 1
favChar of Harry is c
salary of Harry is 120000000
```

Chapter 25 - Unions

Just like Structures, the union is a user-defined data type. They provide better memory management than structures. All the members in the unions share the same memory location.

The union is a data type that allows different data belonging to different data types to be stored in the same memory locations. One of the advantages of using a union over structures is that it provides an efficient way of reusing the memory location, as only one of its members can be accessed at a time. A union is used in the same way we declare and use a structure. The difference lies just in the way memory is allocated to their members.

Creating a Union element

We use the union keyword to define the union.

The syntax for defining a union is,

```
union union_name
{
    //union_elements
} union_variable;
```

Here's one example of how a union is defined and used in main as a user-defined data type.

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

union money
{
    /* data */
    int rice;
    char car;
    float pounds;
};

int main()
{
    union money m1;
}
```

Initialising and accessing union elements

Different from how we used to initialise a struct in one single statement, union elements are initialised one at a time.

And also, one can access only one union element at a time. Altering one union element disturbs the value stored in other union elements.

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

union money
{
    /* data */
    int rice;
    char car;
    float pounds;
};

int main()
{
    union money m1;
    m1.rice = 34;
    cout << m1.rice;
    return 0;
}
```

Output:

```
34
```

> **Note**: We can only use 1 variable at a time otherwise the compiler will give us a garbage value and the compiler chooses the data type which has maximum memory for the allocation.

Chapter 26 - Enums

Enum or enumeration is a user-defined data type. Enums have named constants that represent integral values. Enums are used to make the program more readable and less complex. It lets us define a fixed set of possible values and later define variables having one of those values.

Creating an Enum element

We use the enum keyword to define the enum.

The syntax for defining a union is,

```
enum enum_name
{
    element1,
    element2,
    element3
};
```

Here's one example of how a union is defined and used in main as a user-defined data type.

```
enum Meal
{
    breakfast,
    lunch,
    dinner
};
```

Initialising and using enum elements

Since every enum element gets assigned a value to it, they could be used to compare if a particular variable store the same value.

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

enum Meal
{
    breakfast,
    lunch,
    dinner
};

int main()
{
    Meal m1 = dinner;
```

```
    if (m1 == 2)
    {
        cout << "The value of dinner is " << dinner << endl;
    }
}
```

Output:

```
The value of dinner is 2
```

Chapter 27 - Functions

Functions are the main part of top-down structured programming. We break the code into small pieces and make functions of that code. Functions could be called multiple or several times to provide reusability and modularity to the C++ program.

Functions are also called procedures or subroutines or methods and they are often defined to perform a specific task. And that makes functions a group of code put together and given a name that can be called anytime without writing the whole code again and again in a program.

Advantages of Functions

- The use of functions allows us to avoid re-writing the same logic or code over and over again.
- With the help of functions, we can divide the work among the programmers.
- We can easily debug or can find bugs in any program using functions.
- They make code readable and less complex.

Aspects of a function

- **Declaration:** This is where a function is declared to tell the compiler about its existence.
- **Definition:** A function is defined to get some task executed. (It means when we define a function, we write the whole code of that function and this is where the actual implementation of the function is done).
- **Call:** This is where a function is called in order to be used.

Function Prototype in C++

The function prototype is the template of the function which tells the details of the function which include its name and parameters to the compiler. Function prototypes help us to define a function after the function call.

Example of a function prototype,

```
// Function prototype
return_datatype function_name(datatype_1 a, datatype_2 b);
```

Types of functions

Library functions:

Library functions are pre-defined functions in C++ Language. These are the functions that are included in C++ header files prior to any other part of the code in order to be used.

E.g. sqrt(), abs(), etc.

User-defined functions

User-defined functions are functions created by the programmer for the reduction of the complexity of a program. Rather, these are functions that the user creates as per the requirements of a program.

E.g. Any function created by the programmer.

Chapter 28 - Functions Parameters

A function receives information that is passed to them as a parameter. Parameters act as variables inside the function.

Parameters are specified collectively inside the parentheses after the function name. parameters inside the parentheses are comma separated.

We have different names for different parameters.

Formal Parameters

So, the variable which is declared in the function is called a formal parameter or simply, a parameter. For example, variables a and b are formal parameters.

```
int sum(int a, int b){
   //function body
}
```

Actual Parameters

The values which are passed to the function are called actual parameters or simply, arguments. For example, the values num1 and num2 are arguments.

```
int sum(int a, int b);

int main()
{
    int num1 = 5;
    int num2 = 6;
    sum(num1, num2);//actual parameters
}
```

A function doesn't need to have parameters or it should necessarily return some value.

Chapter 29 - Methods

Now, there are methods using which arguments are sent to the function. They are,

Call by Value in C++

Call by value is a method in C++ to pass the values to the function arguments. In the case of call by value the copies of actual parameters are sent to the formal parameter, which means that if we change the values inside the function that will not affect the actual values.

An example that demonstrates the call by value method is,

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

void swap(int a, int b)
{
    int temp = a;
    a = b;
    b = temp;
}

int main()
{
    int x = 5, y = 6;
    cout << "The value of x is " << x << " and the value of y is " << y << endl;
    swap(x, y);
    cout << "The value of x is " << x << " and the value of y is " << y << endl;
}
```

Output:

```
The value of x is 5 and the value of y is 6
The value of x is 5 and the value of y is 6
```

Call by Pointer in C++

A call by the pointer is a method in C++ to pass the values to the function arguments. In the case of call by pointer, the address of actual parameters is sent to the formal parameter, which means that if we change the values inside the function that will affect the actual values.

An example that demonstrates the call by pointer method is,

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

```
void swap(int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main()
{
    int x = 5, y = 6;
    cout << "The value of x is " << x << " and the value of y is " << y << endl;
    swap(&x, &y);
    cout << "The value of x is " << x << " and the value of y is " << y << endl;
}
```

Output:

```
The value of x is 5 and the value of y is 6
The value of x is 6 and the value of y is 5
```

Call by Reference in C++

Call by reference is a method in C++ to pass the values to the function arguments. In the case of call by reference, the reference of actual parameters is sent to the formal parameter, which means that if we change the values inside the function that will affect the actual values.

An example that demonstrates the call by reference method is,

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

void swap(int &a, int &b)
{
    int temp = a;
    a = b;
    b = temp;
}

int main()
{
    int x = 5, y = 6;
    cout << "The value of x is " << x << " and the value of y is " << y << endl;
    swap(x, y);
    cout << "The value of x is " << x << " and the value of y is " << y << endl;
}
```

Output:

```
The value of x is 5 and the value of y is 6
The value of x is 6 and the value of y is 5
```

Two special methods that are often used by programmers to have their program work efficiently are,

Default Arguments in C++

Default arguments are those values which are used by the function if we don't input our value as parameter. It is recommended to write default arguments after the other arguments.

An example using default argument,

```
int sum(int a = 5, int b);
```

Constant Arguments in C++

Constant arguments are used when you don't want your values to be changed or modified by the function. The const keyword is used to make the parameters non-modifiable.

An example using constant argument,

```
int sum(const int a, int b);
```

Chapter 30 - Recursion

When a function calls itself, it is called recursion and the function which is calling itself is called a recursive function. The recursive function consists of a base condition and a recursive condition.

Recursive functions must be designed with a base case to make sure the recursion stops, otherwise, they are bound to execute forever and that's not what you want. The case in which the function doesn't recur is called the base case. For example, when we try to find the factorial of a number using recursion, the case when our number becomes smaller than 1 is the base case.

An example of a recursive function is the function for calculating the factorial of a number.

```
int factorial(int n){
    if (n == 0 || n == 1){
        return 1;
    }
    return n * factorial(n - 1);
}
```

Chapter 31 - Function Overloading

Function overloading is a process to make more than one function with the same name but different parameters, numbers, or sequences. Now, there are a few conditions and any number of functions with the same name following any of these are called overloaded.

Same name but different data type of parameters

Example

```
float sum(int a, int b);
float sum(float a, float b);
```

Same name but a different number of parameters

Example

```
float sum(int a, int b);
float sum(int a, int b, int c);
```

Same name but different parameter sequence

Example

```
float sum(int a, float b);
float sum(float a, int b);
```

Exact matches are always preferred while looking for a function that has the same set of parameters.

Chapter 32 - C++ OOP Basics

What is OOP?

OOP stands for Object-Oriented Programming. An object-oriented programming language uses objects in its programming. Programming with object-oriented concepts aims to emulate real-world concepts such as inheritance, polymorphism, abstraction, etc, in a program.

C++ language was designed with the main intention of adding object-oriented programming to C language. As the size of the program increases, the readability, maintainability, and bug-free nature of the program decrease. The main aim of OOP is to bind together the data and the functions that operate on them so that no other part of the code can access this data except that function.

This was the major problem with languages like C which relied upon functions or procedures (hence the name procedural programming language). As a result, the possibility of not addressing the problem adequately was high. Also, data was almost neglected, and data security was easily compromised Using classes solves this problem by modeling the program as a real-world scenario.

Difference between Procedure Oriented Programming and Object-Oriented Programming

Procedure Oriented Programming

- Consists of writing a set of instructions for the computer to follow
- The main focus is on functions and not on the flow of data
- Functions can either use local or global data
- Data moves openly from function to function

Object - Oriented Programming

- Works on the concept of classes and object
- A class is a template to create objects
- Treats data as a critical element
- Decomposes the problem in objects and builds data and functions around the objects

Basically, procedural programming involves writing procedures or functions that manipulate data, while object-oriented programming involves creating objects that contain both data and functions.

Basic Elements in Object-Oriented Programming

- **Classes** - Basic template for creating objects. This is the building block of object-oriented programming.
- **Objects** – Basic run-time entities and instances of a class.
- **Data Abstraction & Encapsulation** – Wrapping data and functions into a single unit
- **Inheritance** – Properties of one class can be inherited into others
- **Polymorphism** – Ability to take more than one form
- **Dynamic Binding** – Code which will execute is not known until the program runs
- **Message Passing** – message (Information) call format

Benefits of Object-Oriented Programming

Object-oriented programming has many advantages. Listed below are a few.

- Programs involving OOP is faster and easier to execute.
- By using objects and inheritance, it provides a clear structure for programs and improves code reusability.
- It makes the code easier to maintain, modify and debug.
- Principle of data hiding helps build secure systems
- Multiple Objects can co-exist without any interference
- Software complexity can be easily managed so that even the creation of fully reusable software with less code and shorter development time is possible.

Chapter 33 - C++ Classes & Objects

Classes

Classes and structures are somewhat the same but still, they have some differences. For example, we cannot hide data in structures which means that everything is public and can be accessed easily which is a major drawback of the structure because structures cannot be used where data security is a major concern. Another drawback of structures is that we cannot add functions to them.

Classes are user-defined data types and are a template for creating objects. Classes consist of variables and functions which are also called class members.

We use the class keyword to define a class in C++.

The syntax of a class in C++ is

```
class class_name
{
    //body of the class
};
```

Objects

Objects are instances of a class. To create an object, we just have to specify the class name and then the object's name. Objects can access class attributes and methods which are bound to the class definition. It is recommended to put these attributes and methods in access modifiers so that their permissions can be better specified to allow them to be used by objects.

The syntax for defining an object in C++ is

```
class class_name
{
    //body of the class
};

int main()
{
    class_name object_name; //object
}
```

Chapter 34 - Class Attributes & Methods

Class attributes and methods are variables and functions that are defined inside the class. They are also known as class members altogether.

Consider an example below to understand what class attributes are

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

class Employee
{
    int eID;
    string eName;
    public:
};


int main()
{
    Employee Harry;
}
```

A class named Employee is built and two members, eId and eName are defined inside the class. These two members are variables and are known as class attributes. Now, an object named Harry is defined in the main. Harry can access these attributes using the dot operator. But they are not accessible to Harry unless they are made public.

```
class Employee
{
public:
    int eID;
    string eName;
};


int main()
{
    Employee Harry;
    Harry.eID = 5;
    Harry.eName = "Harry";
    cout << "Employee having ID " << Harry.eID << " is " << Harry.eName << endl;
}
```

Output:

```
Employee having ID 5 is Harry
```

Class methods are nothing but functions that are defined in a class or belong to a class. Methods belonging to a class are accessed by their objects in the same way that they access attributes. Functions can be defined in two ways so that they belong to a class.

Defining inside the class

An example that demonstrates defining functions inside classes is

```
class Employee
{
public:
    int eID;
    string eName;

    void printName()
    {
        cout << eName << endl;
    }
};
```

Defining outside the class

Although, a function can be defined outside the class, it needs to be declared inside. Later, we can use the scope resolution operator (::) to define the function outside.

An example that demonstrates defining functions outside classes is

```
class Employee
{
public:
    int eID;
    string eName;

    void printName();
};


void Employee::printName()
{
    cout << eName << endl;
}
```

Chapter 35 - Objects Memory Allocation

Objects Memory Allocation in C++

The way memory is allocated to variables and functions of the class is different even though they both are from the same class. The memory is only allocated to the variables of the class when the object is created.

The memory is not allocated to the variables when the class is declared. At the same time, single variables can have different values for different objects, so every object has an individual copy of all the variables of the class. But the memory is allocated to the function only once when the class is declared. So the objects don't have individual copies of functions only one copy is shared among each object.

Static Data Members in C++

When a static data member is created, there is only a single copy of the data member which is shared between all the objects of the class. Usually, every object has an individual copy of the attributes unless specified statically.

Static members are not defined by any object of the class. They are exclusively defined outside any function using the scope resolution operator.

An example of how static variables are defined is

```
class Employee
{

public:
    static int count; //returns number of employees
    string eName;

    void setName(string name)
    {
        eName = name;
        count++;
    }
};

int Employee::count = 0; //defining the value of count
```

Static Methods in C++

When a static method is created, they become independent of any object and class. Static methods can only access static data members and static methods. Static methods can only be accessed using the scope resolution operator. An example of how static methods are used in a program is shown.

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

```
class Employee
{

public:
    static int count; //static variable
    string eName;

    void setName(string name)
    {
        eName = name;
        count++;
    }

    static int getCount()//static method
    {
        return count;
    }
};

int Employee::count = 0; //defining the value of count

int main()
{
    Employee Harry;
    Harry.setName("Harry");
    cout << Employee::getCount() << endl;
}
```

Output:

```
1
```

Chapter 36 - Friend Functions & Classes

Friend functions are those functions that have the right to access the private data of members of the class even though they are not defined inside the class. It is necessary to write the prototype of the friend function.

Declaring a friend function inside a class does not make that function a member of the class.

Properties of Friend Function

- Not in the scope of the class, means it is not a member of the class.
- Since it is not in the scope of the class, it cannot be called from the object of that class.
- Can be declared anywhere inside the class, be it under the public or private access modifier, it will not make any difference
- It cannot access the members directly by their names, it needs (object_name.member_name) to access any member.

The syntax for declaring a friend function inside a class is

```
class class_name
{
    friend return_type function_name(arguments);
};

return_type class_name::function_name(arguments)
{
    //body of the function
}
```

Friend Classes in C++

Friend classes are those classes that have permission to access private members of the class in which they are declared. The main thing to note here is that if the class is made friends of another class then it can access all the private members of that class.

The syntax for declaring a friend class inside a class is

```
class class_name
{
    friend class friend_class_name;
};
```

Chapter 37 - C++ Constructors

A constructor is a special member function with the same name as the class. The constructor doesn't have a return type. Constructors are used to initialize the objects of their class. Constructors are automatically invoked whenever an object is created.

Characteristics of Constructors in C++

- A constructor should be declared in the public section of the class.
- They are automatically invoked whenever the object is created.
- They cannot return values and do not have return types.
- It can have default arguments.

An example of how a constructor is used is,

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

class Employee
{

public:
    static int count; //returns number of employees
    string eName;

    //Constructor
    Employee()
    {
        count++; //increases employee count every time an object is defined
    }

    void setName(string name)
    {
        eName = name;
    }

    static int getCount()
    {
        return count;
    }
};

int Employee::count = 0; //defining the value of count

int main()
{
    Employee Harry1;
    Employee Harry2;
    Employee Harry3;
```

```
    cout << Employee::getCount() << endl;
}
```

Output:

```
3
```

Parameterized and Default Constructors in C++

Parameterized constructors are those constructors that take one or more parameters. Default constructors are those constructors that take no parameters. This could have helped in the above example by passing the employee name at the time of definition only. That should have removed the setName function.

Constructor Overloading in C++

Constructor overloading is a concept similar to function overloading. Here, one class can have multiple constructors with different parameters. At the time of definition of an instance, the constructor, which will match the number and type of arguments, will get executed.

For example, if a program consists of 3 constructors with 0, 1, and 2 arguments and we pass just one argument to the constructor, the constructor which is taking one argument will automatically get executed.

Constructors with Default Arguments in C++

Default arguments of the constructor are those which are provided in the constructor declaration. If the values are not provided when calling the constructor the constructor uses the default arguments automatically.

An example that shows declaring default arguments is

```
class Employee
{

public:
    Employee(int a, int b = 9);
};
```

Copy Constructor in C++

A copy constructor is a type of constructor that creates a copy of another object. If we want one object to resemble another object we can use a copy constructor. If no copy constructor is written in the program compiler will supply its own copy constructor.

The syntax for declaring a copy constructor is

```
class class_name
{
```

```
    int a;

public:
    //copy constructor
    class_name(class_name &obj)
    {
        a = obj.a;
    }
};
```

Chapter 38 - C++ Encapsulation

Encapsulation is the first pillar of Object Oriented Programming. It means wrapping up data attributes and methods together. The goal is to keep sensitive data hidden from users.

Encapsulation is considered a good practice where one should always make attributes private for them to become non-modifiable until needed. The data is ultimately more secure as a result of this. Once members are made private, methods to access them or change them should be declared.

An example of how encapsulation is achieved is

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

class class_name
{
private:
    int a;

public:
    void setA(int num)
    {
        a = num;
    }

    int getA()
    {
        return a;
    }
};

int main()
{
    class_name obj;
    obj.setA(5);
    cout << obj.getA() << endl;
}
```

Output:

```
5
```

Chapter 39 - C++ Inheritance

What is Inheritance in C++?

The concept of reusability in C++ is supported using inheritance. We can reuse the properties of an existing class by inheriting it and deriving its properties. The existing class is called the base class and the new class which is inherited from the base class is called the derived class.

The syntax for inheriting a class is

```
// Derived Class syntax
class derived_class_name : access_modifier base_class_name
{
    // body of the derived class
}
```

Types of inheritance in C++

Single Inheritance

Single inheritance is a type of inheritance in which a derived class is inherited with only one base class.

For example, we have two classes ClassA and ClassB. If ClassB is inherited from ClassA, it means that ClassB can now implement the functionalities of ClassA. This is single inheritance.

```
class ClassA
{
    //body of ClassA
};

//derived from ClassA
class ClassB : public ClassA
{
    //body of ClassB
};
```

Multiple Inheritance

Multiple inheritances is a type of inheritance in which one derived class is inherited from more than one base class.

For example, we have three classes ClassA, ClassB, and ClassC. If ClassC is inherited from both ClassA & ClassB, it means that ClassC can now implement the functionalities of both ClassA & ClassB. This is multiple inheritances.

```
class ClassA
{
    //body of ClassA
};

class ClassB
{
    //body of ClassB
};

//derived from ClassB and Class C
class ClassC : public ClassA, public ClassB
{
    //body of ClassC
};
```

Hierarchical Inheritance

A hierarchical inheritance is a type of inheritance in which several derived classes are inherited from a single base class.

For example, we have three classes ClassA, ClassB, and ClassC. If ClassB and Class C are inherited from ClassA, it means that ClassB and ClassC can now implement the functionalities of ClassA. This is hierarchical inheritance.

```
class ClassA
{
    //body of ClassA
};

//derived from ClassA
class ClassB : public ClassA
{
    //body of ClassB
};

//derived from ClassA
class ClassC : public ClassA
{
    //body of ClassC
};
```

Multilevel Inheritance

Multilevel inheritance is a type of inheritance in which one derived class is inherited from another derived class.

For example, we have three classes ClassA, ClassB, and ClassC. If ClassB is inherited from ClassA and ClassC is inherited from ClassB, it means that ClassB can now implement the functionalities of ClassA and ClassC can now implement the functionalities of ClassB. This is multilevel inheritance.

```
class ClassA
{
    //body of ClassA
};

//derived from ClassA
class ClassB : public ClassA
{
    //body of ClassB
};

//derived from ClassB
class ClassC : public ClassB
{
    //body of ClassC
};
```

Hybrid Inheritance

Hybrid inheritance is a combination of different types of inheritances.

For example, we have four classes ClassA, ClassB, ClassC, and ClassD. If ClassC is inherited from both ClassA and ClassB and ClassD is inherited from ClassC, it means that ClassC can now implement the functionalities of both ClassA and ClassB and ClassD can now implement the functionalities of ClassC. This is multilevel inheritance where both multilevel and multiple inheritances are present.

```
class ClassA
{
    //body of ClassA
};

class ClassB
{
    //body of ClassB
};

//derived from ClassA and ClassA
class ClassC : public ClassA, public ClassB
{
    //body of ClassC
};

//derived from ClassC
class ClassD : public ClassC
{
```

```
    //body of ClassD
};
```

Chapter 40 - C++ Access Modifiers

Public Access Modifier in C++

All the variables and functions declared under the public access modifier will be available for everyone. They can be accessed both inside and outside the class. Dot (.) operator is used in the program to access public data members directly.

Private Access Modifier in C++

All the variables and functions declared under a private access modifier can only be used inside the class. They are not permissible to be used by any object or function outside the class.

Protected Access Modifiers in C++

Protected access modifiers are similar to the private access modifiers but protected access modifiers can be accessed in the derived class whereas private access modifiers cannot be accessed in the derived class.

A table is shown below which shows the behavior of access modifiers when they are derived from public, private, and protected.

	Public derivation	**Private Derivation**	**Protected Derivation**
Private members	Not inherited	Not inherited	Not inherited
Protected Members	Protected	Private	Protected
Public Members	Public	Private	Protected

- If the class is inherited in public mode then its private members cannot be inherited in child class.
- If the class is inherited in public mode then its protected members are protected and can be accessed in child class.
- If the class is inherited in public mode then its public members are public and can be accessed inside the child class and outside the class.
- If the class is inherited in private mode then its private members cannot be inherited in child class.
- If the class is inherited in private mode then its protected members are private and cannot be accessed in child class.
- If the class is inherited in private mode then its public members are private and cannot be accessed in child class.
- If the class is inherited in protected mode then its private members cannot be inherited in child class.
- If the class is inherited in protected mode then its protected members are protected and can be accessed in the child class.
- If the class is inherited in protected mode then its public members are protected and can be accessed in the child class.

Chapter 41 - C++ Polymorphism

Polymorphism in C++

Poly means several and morphism means form. Polymorphism is something that has several forms or we can say it as one name and multiple forms. There are two types of polymorphism:

- Compile-time polymorphism
- Run time polymorphism

Compile Time Polymorphism

In compile-time polymorphism, it is already known which function will run. Compile-time polymorphism is also called early binding, which means that you are already bound to the function call and you know that this function is going to run.

There are two types of compile-time polymorphism:

Function Overloading

This is a feature that lets us create more than one function and the functions have the same names but their parameters need to be different. When function overloading is implemented in a program and function calls are made, the compiler knows which functions to execute.

Operator Overloading

This is a feature that lets us define operators working for some specific tasks. Using the operator +, we can add two strings by concatenating and add two numerical values arithmetically at the same time. This is operator overloading.

Run Time Polymorphism

With run-time polymorphism, the compiler has no idea what will happen when the code is executed. Run time polymorphism is also called late binding. The run-time polymorphism is considered slow because function calls are decided at run time. Run time polymorphism can be achieved from virtual functions.

Virtual Functions in C++

A member function in the base class that is declared using a virtual keyword is called a virtual function. They can be redefined in the derived class. A function that is in the parent class but redefined in the child class is called a virtual function.

For a function to become virtual, it should not be static.

Chapter 42 - C++ File I/O

The file is a patent of data stored on the disk. Anything written inside the file is called a patent, for example: “#include” is a patent. The text file is the combination of multiple types of characters, for example, the semicolon “;” is a character.

The computer read these characters in the file with the help of the ASCII code. Every character is mapped on some decimal number. For example, the ASCII code for the character A is 65 which is a decimal number. These decimal numbers are converted into binary numbers to make them readable for the computer because the computer can only understand the language of 0 & 1.

A large program deployed for heavy applications cannot function without files, since we can get input from them as well as print output from them very easily. We can also save a lot of program space by accessing the file's data only when needed, making the program more efficient and faster.

Files are stored in non-volatile memory which is better in terms of storing data. Non-volatile memory stays intact even after the system shuts down. They get retrieved again after the system is turned on.

Operations on files

There are basically four operations we can perform on files in C.

Creating a File:

We can create a file using C++ language, in any directory, without even leaving our compiler. We can select the name or type we want our file to have, along with its location.

Opening a File:

We can open an existing file and create a new file and open it using our program. We can perform different operations on a file after it has been opened.

Closing a File:

When we are done with the file, meaning that we have performed whatever we want to perform on our file, we can close the file using the close function.

Read/Write to a file:

After opening a file, we can access its contents and read, write, or update them.

Chapter 43 - File I/O Functions

These are some useful classes for working with files in C++

- fstream - A combination of ofstream and ifstream: creates, reads, and writes to files
- ofstream - creates and writes to files
- ifstream - reads from files

The fstream library allows us to handles files. So, to be able to use files in a program, one must include the `<fstream>` header file.

A. Opening a file

In order to work with files in C++, we will first have to open it. Primarily, there are two ways to open a file:

- Using the constructor
- Using the member function open() of the class

A file could be opened for a number of uses. It could be to write to it or to read from it.

Consider an example that demonstrates the opening of a file using a constructor.

```
#include <iostream>
#include <fstream>

using namespace std;

int main()
{
    ofstream out("example.txt");
    return 0;
}
```

The file example.txt gets created if it's not already there in the system and opened. Object out gets created of the type ofstream, which means it could only be used to write into the opened file.

This is how we use the constructor ofstream to open a file. Another example demonstrates the use of the ifstream constructor.

```
#include <iostream>
#include <fstream>

using namespace std;

int main()
{
    ifstream in("example.txt");
```

```
    return 0;
}
```

For this to work, a file named example.txt must already be created and present in the same folder as that of the program. Object in gets created of the type ifstream, which means it could only be used to read from the file.

B. Closing a file

When working with C++, closing open files is considered a good practice. A programmer often makes the mistake of not closing an open file. This becomes crucial because files do not automatically get closed after a program uses them. The closing has to be done manually.

To close a file, we have to use the close method. This is how we use them in C++.

Syntax:

```
file_objectname.close();
```

C. Writing to a file

Writing to a file is as easy as printing any other stuff in C++. It is very similar to what we used to do when we had to print an output in the terminal. In order to write to a file, we use the insertion operator (<<). First, we create an object of the type ofstream and pass the name of the file along with its extension to the method. And then, use the extraction operator to write stuff in the file fed to the object.

Consider an example demonstrating how we write to a file.

Example:

```
#include <iostream>
#include <fstream>

using namespace std;

int main()
{
    string str = "Hello World!";
    ofstream out("example.txt");
    out << str;
    return 0;
}
```

Output in the example.txt file:

```
Hello World!
```

D. Reading a file

Reading from a file is as easy as reading any other stuff as an input in C++. It is very similar to what we used to do when we had to read input from the terminal. In order to read from a file, we use the extraction operator (>>). First, we create an object of the type ifstream and pass the name of the file along with its extension to the method. And then, use the extraction operator to read stuff from the file fed to the object.

Consider an example demonstrating how we read from a file.

Example:

```
#include <iostream>
#include <fstream>

using namespace std;

int main()
{
    string str;
    ifstream in("example.txt");
    in >> str;
    cout << str;
    return 0;
}
```

The file example.txt had “Hello World!” as its content, hence the output:

```
Hello World!
```

Chapter 44 - C++ Templates

What are templates?

A template is believed to escalate the potential of C++ several fold by giving it the ability to define data types as parameters making it useful to reduce repetitions of the same declaration of classes for different data types.

Declaring classes for every other data type doesn't completely utilize the potential of C++. It is very analogous to when we said classes are the templates for objects, here templates themselves are the templates of the classes. That is, what classes are for objects, templates are for classes.

Why do we use templates?

To understand the reason behind using templates, we will have to understand the effort behind declaring classes for different data types. Suppose we want to have a structure for each of the three data types, int, float, and char. Then we'll obviously write the whole thing again and again making it awfully difficult. This is where the savior comes, the templates. It helps to parametrize the data type and then, declare it once in the source code suffice. Very similar to what we do in functions. It is because of this, also called, parameterized classes.

Syntax of a template

Consider a class containing two elements, an integer pointer, and an integer variable size. This class acts like an array of integers with a given length size. We often name them a vector.

```
class vector
{
    int *arr;
    int size;
};
```

But, if we need another array of some other data type, we will have to redefine another class for the same. To avoid that repetition of logic, templates help. Here's how we do subtle changes in the same definition of the class for an integer array to make it generic for any other data type as well.

```
template <class T>
class vector
{
    T *arr;
    int size;
};
```

Here, we declared a template of class and passed a variable T as its parameter. Then, we defined the class of vector and keep the data type of *arr as T only. Now, the array becomes of the type we supply in the template

as T.

So, the same vector can now be used as an integer array as well as a float or character array.

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

template <class T>
class vector
{
    T *arr;
    int size;
};

int main()
{
    vector<int> v1();
    vector<float> v2();
}
```

Chapter 45 - C++ Class Templates

Class templates are one application of templates. Class templates could be used according to the requirements of the programmer. Templates are nothing but shortened generic declarations of similar entities. They are expanded at the compile time.

Examples of some of the in-built class templates are,

Vector, Set, Linked List etc.

Class templates are helpful for classes that are independent of the data type. And like any other function, there could be more than one parameter to a template. On the basis of the number of parameters of a template, they could be single parameters as well as multiple parameters.

A. Single parameter

Syntax for declaring a single parametrized template is,

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

template <class T>
class nameOfClass
{
    //body
};

int main()
{
    //body of main
}
```

The example of the vector easily demonstrates how templates are used with a single parameter.

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

template <class T>
class vector
{
    T *arr;
    int size;
};

int main()
{
    vector<int> v1();
    vector<float> v2();
}
```

B. Multiple Parameter

Syntax for declaring a multiple parametrized template is,

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

template <class T1, class T2>
class nameOfClass
{
    //body
};

int main()
{
    //body of main
}
```

The difference lies only in the number of parameters we declare inside the template. Consider an example that demonstrates the use of multiple parameters in a class template.

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

template <class T1, class T2>

class myClass
{
public:
    T1 data1;
    T2 data2;
    myClass(T1 a, T2 b)
    {
        data1 = a;
        data2 = b;
    }
    void display()
    {
        cout << this->data1 << " " << this->data2;
    }
};

int main()
{
    myClass<char, int> obj('C', 1);
    obj.display();
}
```

Output:

```
C 1
```

Now, we defined an object obj in myClass to hold a character and an integer data. It could be anything else since both the parameters have been generalized using the template.

C. Default Parameter

C++ Templates have this additional ability to have default parameters. Its ability to have default specifications about the data type, when it receives no arguments from the main is a powerful attribute.

The syntax for making a parameter have a default datatype,

```
template <class T1 = datatype_1, class T2 = datatype_2>
```

Now, even if the programmer does not specify the type of the data, objects created using this template will automatically make it, of the form `<datatype_1, datatype_2>`.

Consider an example demonstrating the application of the default parameters,

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

template <class T1 = int, class T2 = float, class T3 = char>
class myClass
{
public:
    T1 data1;
    T2 data2;
    T3 data3;

    myClass(T1 a, T2 b, T3 c)
    {
        data1 = a;
        data2 = b;
        data3 = c;
    }
    void display()
    {
        cout << "The value of a is " << data1 << endl;
        cout << "The value of b is " << data2 << endl;
        cout << "The value of c is " << data3 << endl;
    }
};

int main()
{
```

```
    myClass<> obj1(1, 4.3, 'C');
    obj1.display();
}
```

Output:

```
The value of a is 1
The value of b is 4.3
The value of c is C
```

The template has default defined its parameters as an integer, a float, and a character. Now, if the programmer wishes to change the data types, it must be exclusively mentioned and the object must be defined as it was done in earlier examples.

Chapter 46 - C++ Function Templates

Function templates are another application of templates. Class templates were used to program generic classes. Similarly, function templates are used to define generic functions to be able to use a function independent of its data types for different data type combinations.

Example of some of the in-built function templates are,

sort(), max(), min() etc.

The syntax for constructing a function template is,

```
template <class T1, class T2>
data_type function_name(T1 a, T2 b)
{
    //function body
}
```

An example of a somewhat data type independent function might be a function to find the average of two numerical values. Now, this function might expect any of the data type combinations between an integer, a float, or a double value. So, here is how we build a function so that it is generalized for finding the average of two numerical values.

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

template <class T1, class T2>
float findAverage(T1 a, T2 b)
{
    float avg = (a + b) / 2.0;
    return avg;
}

int main()
{
    float avg = findAverage(5.1, 2);
    cout << avg << endl;
}
```

Output:

```
3.55
```

Both class templates and function templates help reduce the size of the program and make it more readable.

Chapter 47 - C++ STL

What is Standard Template Library?

Competitive programming is a part of various environments, be it job interviews, coding contests, and all, and if you're in one of those environments, you'll be given limited time to code your program. So, suppose you want in your program, a resizable array, sort an array or any other data structure. or search for some element in your container. You will always try to code a function that will execute the above-mentioned things, and end up losing a great amount of time. But here is when you will use STL.

An STL is a library of generic functions and classes which saves you time and energy which you would have spent constructing for your use. This helps you reuse these well-tested classes and functions an umpteenth number of times according to your own convenience. To put this simply, STL is used because it is not a good idea to reinvent something which is already built and can be used to innovate things further. Suppose you go to a company that builds cars, they will not ask you to start from scratch, but to start from where it is left. This is the basic idea behind using STL.

What are the components of STL?

- **Containers**

  Container is an object that stores data. We have different containers having their own benefits. These are the implemented class templates for our use, which can be used just by including this library. You can even customize these template classes.

- **Algorithm**

  Algorithms are sets of instructions that manipulate the input data to arrive at some desired result. In STL, we have already written algorithms, for example, to sort some data structure, or search some element in an array. These algorithms use function templates.

- **Iterators**

  Iterators are objects which refer to an element in a container. And we handle them very much similar to a pointer. Their basic job is to connect algorithms to the container and play a vital role in the manipulation of the data.

Chapter 48 - C++ Containers

As discussed earlier, containers are objects that store information and they are nothing but class templates with specific functionalities. Containers themselves are of three types:

Sequence Containers

A sequence container stores that data in a linear fashion. Sequence containers include Vector, List, Dequeue, etc. These are some of the most used sequence containers.

Associative Containers

An associative container is designed in such a way that enhances the accessing of some element in that container. It is very much used when the user wants to fastly reach some element. Some of these containers are, Set, Multiset, Map, Multimap etc. They store their data in a tree-like structure.

Derived Containers

As the name suggests, these containers are derived from either the sequence or the associative containers. They often provide you with some better methods to deal with your data. They deal with real life modeling. Some examples of derived containers are Stack, Queue, Priority Queue, etc.

Chapter 49 - C++ Vectors

A. What are vectors?

Vectors are sequence containers that fall under the category of class templates. They are used to store information in a linear fashion. Elements of a single data type can be stored in a vector. In a vector, we can access elements faster in comparison to arrays. Although, insertion and deletion at some random position, except at the end is comparatively slower. And inserting any element at the end happens faster.

B. Using a vector in our programs

To be able to use vectors in our code, the header file `<vector>` must be included. And the syntax to define a vector is

```
vector<data_type> vector_name;
```

The data_type could be replaced by any data type. One benefit of using vectors is that we can insert as many elements as we want in a vector, without having to put some size parameter as we do in an array.

Vectors provide certain methods to be used to access and utilize the elements of a vector, the first one being, the push_back method. To access all the methods and member functions in detail, one can visit this site, std::vector - C++ Reference.

C. Initialising a vector

A vector could be initialized in different ways:

- In the first method, a vector could be initialized with all the elements inserted in the vector at the time it is defined.

  ```
  vector<int> v = {1, 2, 3, 4};
  ```

- Another method to initialize a vector is where we pass two parameters along with its definition where the first parameter defines the size of the vector and the second parameter is the uniform value it will have at all its positions.

  ```
  vector<int> v(4, 10);
  ```

  Here, v is of size 4 with value 10 at all positions.

D. Inserting elements in a vector

We can insert elements in a vector in different ways. The most economical way to insert an element in a vector is when we insert it at the rear end using the push_back method. The push_back() function gets the elements to be inserted as a parameter and the element gets pushed at the back.

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

void display(vector<int> v)
{
    cout << "The elements are: ";
    for (auto it : v)
    {
        cout << it << " ";
    }
    cout << endl;
}

int main()
{
    vector<int> v = {1, 2, 3, 4};
    display(v);
    v.push_back(5); //5 is inserted at the back
    display(v);
}
```

Output:

```
The elements are: 1 2 3 4
The elements are: 1 2 3 4 5
```

Another method to insert an element allows us to insert at any random position. Here, we use the insert() method. The syntax for using the insert method is

```
vector_name.insert(iterator, element);
```

Here, the iterator is the pointer to that position where the element gets inserted. An example that demonstrates the use of the insert() method is

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

void display(vector<int> v)
{
```

```
    cout << "The elements are: ";
    for (auto it : v)
    {
        cout << it << " ";
    }
    cout << endl;
}

int main()
{
    vector<int> v = {1, 2, 3, 4};
    display(v);
    v.insert(v.begin(), 5);
    display(v);
}
```

Output:

```
The elements are: 1 2 3 4
The elements are: 5 1 2 3 4
```

Here, we used the begin() method which returned an iterator to the first position of the vector. We can manipulate it accordingly to point at the desired position by general arithmetic operations.

E. Accessing/Changing elements in a vector

Changing any element at any position is similar to accessing them. Any element at any specified position could be accessed using its index number as it was done for arrays.

```
vector<int> v = {1, 2, 3, 4};
cout << "Element at index 0 is " << v[0] << endl;
```

Output:

```
Element at index 0 is 1
```

Another method that gets added for vectors is the at() method which accepts the index number as its parameter and returns the element at that position.

```
vector<int> v = {1, 2, 3, 4};
cout << "Element at index 2 is " << v.at(2) << endl;
```

Output:

```
Element at index 2 is 3
```

F. Removing elements from a vector

We can remove elements from a vector in different ways. The most economical way to remove an element from a vector is when we remove it from the rear end using the pop_back method. The `pop_back()` function needs nothing as a parameter and the element gets popped from the back.

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

void display(vector<int> v)
{
    cout << "The elements are: ";
    for (auto it : v)
    {
        cout << it << " ";
    }
    cout << endl;
}

int main()
{
    vector<int> v = {1, 2, 3, 4};
    display(v);
    v.pop_back(); //4 gets popped from the back
    display(v);
}
```

Output:

```
The elements are: 1 2 3 4
The elements are: 1 2 3
```

Another method to remove an element allows us to remove it from any random position. Here, we use the erase() method. The syntax for using the erase method is

```
vector_name.erase(iterator);
```

Here, the iterator is the pointer to that position where the element gets popped from. An example that demonstrates the use of the `erase()` method is

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

void display(vector<int> v)
{
    cout << "The elements are: ";
    for (auto it : v)
    {
        cout << it << " ";
    }
    cout << endl;
}

int main()
{
    vector<int> v = {1, 2, 3, 4};
    display(v);
    v.erase(v.begin());
    display(v);
}
```

Output:

```
The elements are: 1 2 3 4
The elements are: 2 3 4
```

Chapter 50 - C++ Lists

A. What are lists?

Lists are sequence containers that fall under the category of class templates. They are also used to store information in a linear fashion. Only elements of a single data type can be stored in a list. In a list, we have faster insertion and deletion of elements as compared to other containers such as arrays and vectors. Although, accessing elements at some random position, is comparatively slower.

B. What makes inserting in and deleting from a list faster?

An array stores the elements in a contiguous manner in which inserting some element calls for a shift of other elements, which is time taking. But in a list, we can simply change the address the pointer is pointing to.

C. Using a list in our programs

To be able to use lists in our code, the header file `<list>` must be included. And the syntax to define a list is

```
list<data_type> list_name;
```

The data_type could be replaced by any data type. One benefit of using lists is that lists support a bidirectional and provide an efficient way for insertion and deletion operations.

Lists provide certain methods to be used to access and utilize the elements of a list, the first one being, the push_back method. To access all the methods and member functions in detail, one can visit this site, std::list.

D. Initialising a list

A list could be initialized in a very similar way we used to initialize a vector.: A list could be initialized with all the elements inserted in the list at the time it is defined.

```
list<int> l = {1, 2, 3, 4};
```

OR it could be even be initialised with elements to be inserted as a parameter.

```
list<int> l{1, 2, 3, 4};
```

E. Inserting elements in a list

We can insert elements in a list in different ways. The insertion of any element in a list takes constant time, making it faster than insertion in vectors or arrays. Insertion in a list could be done using the `push_back()` and the `push_front()` methods. The `push_back()` function gets the elements to be inserted as a parameter

and the element gets pushed at the back. In contrast, the `push_front()` function gets the elements to be inserted as a parameter and the element gets pushed at the front.

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

void display(list<int> l)
{
    cout << "The elements are: ";
    for (auto it : l)
    {
        cout << it << " ";
    }
    cout << endl;
}

int main()
{
    list<int> l = {1, 2, 3, 4};
    display(l);
    l.push_back(5);
    display(l);
    l.push_front(0);
    display(l);
}
```

Output:

```
The elements are: 1 2 3 4
The elements are: 1 2 3 4 5
The elements are: 0 1 2 3 4 5
```

Another method to insert an element allows us to insert at any random position. Here, we use the insert() method. The syntax for using the insert method is

```
list_name.insert(iterator, element);
```

Here, the iterator is the pointer to that position where the element gets inserted. An example that demonstrates the use of the `insert()` method is

```
    list<int> l = {1, 2, 3, 4};
    display(l);
    auto it = l.begin();
    it++;
    it++;
```

```
    l.insert(it, 5); //5 is inserted at the third position
    display(l);
```

Output:

```
The elements are: 1 2 3 4
The elements are: 1 2 5 3 4
```

F. Accessing/Changing elements in a list

Changing or accessing any element in a list is a costlier process in terms of time. Any element at any specified position could not be simply accessed using its index number or anything that we could be done for arrays or vectors.

We have to use the iterators to manually traverse through the list to reach a specific element or position and then dereference the iterator to retrieve the value at that position.

```
list<int> l = {1, 2, 3, 4};
list<int>::iterator it = l.begin();
it++;
it++;
cout << "Element at index 2 is " << *it << endl;
```

Output:

```
Element at index 2 is 3
```

G. Removing elements from a list

We can remove elements from a list in different ways. The removal of any element in a list takes constant time, making it faster than removal in vectors or arrays. Removal in a list could be done using the `pop_back()` and the `pop_front()` methods. The `pop_back()` function needs nothing as a parameter and the element gets popped from the back while the `pop_front()` function pops the element from the front.

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

void display(list<int> l)
{
    cout << "The elements are: ";
    for (auto it : l)
    {
        cout << it << " ";
```

```
    }
    cout << endl;
}

int main()
{
    list<int> l = {1, 2, 3, 4};
    display(l);
    l.pop_back();
    display(l);
    l.pop_front();
    display(l);
}
```

Output:

```
The elements are: 1 2 3 4
The elements are: 1 2 3
The elements are: 2 3
```

Another method to remove an element allows us to remove it from any random position. Here, we use the `erase()` method. The syntax for using the erase method is

```
list_name.erase(iterator);
```

Here, the iterator is the pointer to that position where the element gets erased from. An example that demonstrates the use of the `erase()` method is

```
list<int> l = {1, 2, 3, 4};
display(l);
auto it = l.begin();
it++;
it++;
l.erase(it); //element at index 2 gets erased
display(l);
```

Output:

```
The elements are: 1 2 3 4
The elements are: 1 2 4
```

Chapter 51 - C++ Maps

A. What are maps?

A map in C++ STL is an associative container that stores key-value pairs. To elaborate, a map stores a key of some data type and its corresponding values of some data type. All keys in a map are of a single data type and all values in a map are of a single data type. A map always sorts these key-value pairs by the key elements in ascending order.

B. Using a map in our programs

To be able to use maps in our code, the header file `<map>` must be included. And the syntax to define a map is

```
map<data_type_of_key, data_type_of_value> map_name;
```

The data_type could be replaced by any data type and they could be different for both key and value.

Maps provide certain methods to be used to access and utilize the elements of a map. A few of them are discussed. To access all the methods and member functions in detail, one can visit this site, std::map.

C. Initialising a map

A map could be initialized in a similar way we used to initialize a vector or a list. It is just that it would be pairs of elements and not just single elements while we initialize a map. A map could be initialized with all the key-value pairs inserted in the map at the time it is defined.

```
map<string, int> m = {{“Harry”, 2}, {“Rohan”, 4}};
```

D. Inserting elements in a map

We can insert elements in a map in different ways. The insertion of any element in a map takes logarithmic time, making it faster than insertion in vectors or arrays but still slower than lists.

Insertion in a map could be done using the index operators [ ]. The key is put inside the operator and the value is assigned to it. This key-value pair then gets inserted into the map.

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

void display(map<string, int> m)
{
    cout << "The elements are: " << endl;
    for (auto it : m)
    {
```

```
        cout << it.first << " -> " << it.second << endl;
    }
    cout << endl;
}

int main()
{
    map<string, int> m = {{"Harry", 2}, {"Rohan", 4}};
    display(m);
    m["Coder"] = 3;
    display(m);
}
```

Output:

```
The elements are:
Harry -> 2
Rohan -> 4

The elements are:
Coder -> 3
Harry -> 2
Rohan -> 4
```

Another method to insert an element is using the `insert()` method and this allows us to insert as many key-value pairs as we want. The syntax for using the insert method is

```
map_name.insert({key-value pair});
```

An example that demonstrates the use of the `insert()` method is

```
map<string, int> m = {{"Harry", 2}, {"Rohan", 4}};
display(m);
m.insert({{"Coder", 3}, {"Rahul", 5}});
display(m);
```

Output:

```
The elements are:
Harry -> 2
Rohan -> 4

The elements are:
Coder -> 3
Harry -> 2
```

```
Rahul -> 5
Rohan -> 4
```

E. Accessing/Changing elements in a map

Changing or accessing any element in a map is very easy. We just have to use the same key we inserted the value with.

```
map<string, int> m = {{"Harry", 2}, {"Rohan", 4}};
display(m);
m["Harry"] = 1;
display(m);
```

Output:

```
The elements are:
Harry -> 2
Rohan -> 4

The elements are:
Harry -> 1
Rohan -> 4
```

F. Removing elements from a map

We can remove elements from a map in different ways. The removal of any element in a map takes logarithmic time, making it faster than removal in vectors or arrays but slower than lists.

Removal of elements from a map could be done using the `erase()` method. The `erase()` method takes both the keys or an iterator to delete elements. The syntax for using the erase method is

```
map_name.erase(iterator);
//OR
map_name.erase(key);
```

Here, the iterator is the pointer to that position where the element gets erased from and the key is the key of that key-value pair that needs to be removed.

An example that demonstrates the use of the `erase()` method using a key is

```
map<string, int> m = {{"Harry", 2}, {"Rohan", 4}};
display(m);
m.erase("Harry");
display(m);
```

Output:

```
The elements are:
Harry -> 2
Rohan -> 4

The elements are:
Rohan -> 4
```

An example that demonstrates the use of the `erase()` method using a pointer is

```
map<string, int> m = {{"Harry", 2}, {"Rohan", 4}};
display(m);
m.erase(m.begin()); //deletes the first element
display(m);
```

Output:

```
The elements are:
Harry -> 2
Rohan -> 4

The elements are:
Rohan -> 4
```
