/burl@stx null def /BU.S /burl@stx null def def /BU.SS currentpoint /burl@lly

A CONCISE MANUAL ON

C++ PROGRAMMING

Kamakshaiah Musunuru

About the Author

Dr. M. Kamakshaiah is a open source software evangelist, full stack developer and academic of data science and anaytics. His teaching interests are IT practices in business, MIS, Data Science, Business Analytics including functional analytics related to marketing, finance, and HRM, quality and operations. He also teaches theoretical concepts like statistical diagnosis, multivariate analytics, numerical simulations & optimization, machine learning & AI, IoT using few programming languages like R, Python, Java and more.

His research interests are related to few domains such as healthcare, education, food & agriculture, open source software and etc. He has taught abroad for two years and credits two country visits. He has developed few free and open source software solutions meant for corporate practitioners, academics, scholars and students engaging in data science and analytics. All his software applications are available from his GITHUB portal https://H.Bgithub.H.B

com/H.BKamakshaiahH.B.

Contents

1 Introduction **11**

1.1 C vs. C++ . . . 12
1.1.1 What is C? . . . 12
1.1.2 What is C++? . . . 13
1.2 C++ history . . . 15
1.3 C++ Features . . . 15
1.4 Installation . . . 17
1.4.1 Eclipse for C++ . . . 17
1.4.2 Creating C++ project . . . 19
1.5 C++ Program from terminal . . . 19
1.5.1 Object code . . . 21
1.6 Eclipse for C++ development . . . 23
1.7 Basic Input/Output . . . 23
1.7.1 I/O Library Header Files . . . 23
1.7.2 Standard output stream (`cout`) . . . 25
1.7.3 Standard input stream (`cin`) . . . 25
1.7.4 Standard end line (`endl`) . . . 26
1.8 Identifiers and Keywords . . . 26
1.8.1 Identifiers . . . 26
1.8.2 Naming rules . . . 27
1.8.3 Keywords . . . 28
1.9 Data Types . . . 28
1.9.1 Basic Data Types . . . 31
1.9.2 Datatype Modifiers . . . 32
1.10 C++ Variables, Literals and Constants . . . 35
1.10.1 C++ Variables . . . 35

1.10.2 Rules for naming a variable 35
1.10.3 C++ Literals 35
1.10.4 C++ Constants 36
1.11 Operators and Expressions 37
1.11.1 Operators 38
1.11.2 C++ Expression 42

2 Functional programming **55**
2.1 Conditional statements 55
2.1.1 C++ IF Statement 55
2.1.2 C++ IF-else Statement 56
2.1.3 C++ IF-else-if ladder Statement 57
2.2 C++ Switch 59
2.3 C++ For Loop 60
2.3.1 C++ Nested For Loop 60
2.3.2 C++ Infinite For Loop 61
2.4 C++ While loop 61
2.4.1 C++ Nested While Loop Example 62
2.4.2 C++ Infinitive While Loop 63
2.4.3 C++ Do-While Loop 63
2.4.4 C++ Nested do-while Loop 64
2.4.5 C++ Infinitive do-while Loop 65
2.5 C++ Break & Continue Statement 65
2.5.1 C++ Break Statement with Inner Loop . . . 66
2.5.2 C++ Continue Statement 66
2.5.3 C++ Continue Statement with Inner Loop . 67
2.5.4 C++ Goto Statement 68
2.6 Arrays 69
2.6.1 Advantages of C++ Array 69
2.6.2 C++ Array Types 69
2.6.3 One Dimensional Array 70
2.7 Multidimensional arrays 70
2.8 Functions in C/C++ 71
2.8.1 Why do we need functions? 72
2.8.2 Function Declaration 73
2.8.3 Return From Void Functions 73
2.8.4 Parameter Passing to functions 75
2.9 Pointers 77
2.9.1 Advantage of pointer 77

2.9.2 Usage of pointer 78
2.9.3 Symbols used in pointer 78
2.9.4 Declaring a pointer 79
2.9.5 Pointer Example 79
2.9.6 Pointer Expressions and Pointer Arithmetic . 80
2.9.7 References and Pointers 81

3 C++ OOPs Concepts 85
3.0.1 OOPs (Object Oriented Programming System) 85
3.0.2 Advantage of OOPs over Procedure-oriented programming language 86
3.1 C++ Object and Class 87
3.1.1 C++ Class 87
3.1.2 C++ Class Example: Initialize and Display data through method 88
3.1.3 C++ Class Example: Store and Display Employee Information 89
3.2 C++ Constructor 90
3.2.1 C++ Default Constructor 90
3.2.2 C++ Parameterized Constructor 91
3.3 C++ Destructor 91
3.4 C++ this Pointer 93
3.5 C++ static 94
3.5.1 C++ Static Field 94
3.6 C++ Inheritance 96
3.6.1 Types Of Inheritance 96
3.6.2 C++ Single Inheritance 98
3.6.3 C++ Multilevel Inheritance 101
3.6.4 Ambiquity Resolution in Inheritance 103
3.6.5 C++ Hybrid Inheritance 105
3.6.6 C++ Hierarchical Inheritance 107
3.7 C++ Polymorphism 109
3.7.1 Differences b/w compile time and run time polymorphism. 110
3.7.2 Runtime Polymorphism with Data Members . 113

4 File Handling 119
4.1 What is file handling in C++? 119
4.2 The `fstream` Library 119

4.3 How to Open Files 120
4.3.1 Example 1 120
4.4 How to Close Files 121
4.5 Write to Files 121
4.5.1 Example 2 121
4.6 Read from Files 122
4.6.1 Example 3 122

5 Exception Handling **125**
5.1 C++ Exception Classes 125
5.1.1 Accessing Elements of Union 125
5.1.2 C++ Exception Handling Keywords 126
5.2 C++ try/catch 127
5.2.1 C++ example without try/catch 127
5.3 C++ User-Defined Exceptions 128
5.3.1 C++ user-defined exception example 128
5.4 Built-in Exceptions 129
5.4.1 `bad_alloc` exception 129
5.4.2 `bad_cast` exception 130
5.4.3 `bad_typeid` exception 132
5.4.4 `bad_exception` exception 134
5.4.5 `logic_error` exception 136
5.5 Runtime error 137
5.5.1 Invalid memory access 137
5.5.2 Array runs out of bound 138
5.5.3 unassigned variables 138

Preface

This book is a result of my teaching necessity. The necessity arose out of dearth of good text book meant for students in business management stream. There are books on C++ but not sufficient enough to address my class room needs. Most of the students in my classes don't have much of programming background and needs exposure to the same having no compromise on coding.

This text book has five chapters each with a unique goal. This book caters to the needs of both beginners and mavericks. The first chapter deals with introduction to C++ such as history, design principles, few essential components and more. Second chapter deals with few concepts related to code blocks such as data types, operators with related examples. Third chapter deals with control flow consisting code blocks related to conditional statements and loops. The fourth chapter deals with various aspects of Object Oriented Programming (OOP). The last chapter has few code snippets which demonstrates exception handling using C++ in-build methods.

Most of the examples or code blocks are related to data science and analytics such as arithmetic calculations, less intensive numerical analysis etc., so that even a beginner could pick up content without a doubt. There are few code blocks that are brainy and requires sufficient understanding on core statistics such as statistical tests, methods and modes so on.

Happy reading ...

Author

Dr. M. Kamakshaiah

Chapter 1

Introduction

C++ is a general purpose, case-sensitive, free-form programming language that supports object-oriented, procedural and generic programming. C++ is a middle-level language, as it encapsulates both high and low level language features. C++ supports the object-oriented programming, the four major pillar of object-oriented programming (OOPs) used in C++ are:

1. Inheritance
2. Polymorphism
3. Encapsulation
4. Abstraction

Standard C++ programming is divided into three important parts:

1. The core library includes the data types, variables and literals, etc.
2. The standard library includes the set of functions manipulating strings, files, etc.
3. The Standard Template Library (STL) includes the set of methods manipulating a data structure.

By the help of C++ programming language, we can develop different types of secured and robust applications:

1. Window application
2. Client-Server application
3. Device drivers
4. Embedded firmware etc

C++ Example Program

```
#include <iostream>
using namespace std;
int main() {
   cout << "Hello C++ Programming";
   return 0;
}
```

The above program has the sample code that prints a statement `Hello C++ Programming`. The keyword `include` is a C++ keyword which deals with imports. In this program this keyword import one core library called `iostream`. The second statement deals with namespace, which is a common concept in programming. Every program depends on two very important environmental requirements viz., `workspace, namespace`. The other keyword `cout` is `std` command, which takes care of printing output. The last statement `return 0`, is required to tell C++ compiler as what is being returned by the method (`main`, in this case). We will discuss about all these statements in detail through forthcoming sections in this book.

1.1 C vs. C++

1.1.1 What is C?

C is a structural or procedural oriented programming language which is machine-independent and extensively used in various applications. C is the basic programming language that can be used to develop from the operating systems (like Windows) to complex programs like Oracle database, Git, Python interpreter, and many

more. C programming language can be called a god's programming language as it forms the base for other programming languages. If we know the C language, then we can easily learn other programming languages. C language was developed by the great computer scientist *Dennis Ritchie* at the Bell Laboratories. It contains some additional features that make it unique from other programming languages.

1.1.2 What is C++?

C++ is a special-purpose programming language developed by *Bjarne Stroustrup* at Bell Labs circa 1980. C++ language is very similar to C language, and it is so compatible with C that it can run 99% of C programs without changing any source of code though C++ is an object-oriented programming language, so it is safer and well-structured programming language than C.

The following are the differences between C and C++:

1. *Definition*: C is a structural programming language, and it does not support classes and objects, while C++ is an object-oriented programming language that supports the concept of classes and objects.

2. *Type of programming language*: C supports the structural programming language where the code is checked line by line, while C++ is an object-oriented programming language that supports the concept of classes and objects.

3. *Developer of the language*: Dennis Ritchie developed C language at Bell Laboratories while Bjarne Stroustrup developed the C++ language at Bell Labs circa 1980.

4. *Subset*: C++ is a superset of C programming language. C++ can run 99% of C code but C language cannot run C++ code.

5. *Type of approach*: C follows the top-down approach, while C++ follows the bottom-up approach. The top-down approach breaks the main modules into tasks; these tasks are broken into sub-tasks, and so on. The bottom-down approach develops the lower level modules first and then the next level modules.

6. *Security*: In C, the data can be easily manipulated by the outsiders as it does not support the encapsulation and information hiding while C++ is a very secure language, i.e., no outsiders can manipulate its data as it supports both encapsulation and data hiding. In C language, functions and data are the free entities, and in C++ language, all the functions and data are encapsulated in the form of objects.

7. *Function Overloading*: Function overloading is a feature that allows you to have more than one function with the same name but varies in the parameters. C does not support the function overloading, while C++ supports the function overloading.

8. *Function Overriding*: Function overriding is a feature that provides the specific implementation to the function, which is already defined in the base class. C does not support the function overriding, while C++ supports the function overriding.

9. *Reference variables*: C does not support the reference variables, while C++ supports the reference variables.

10. *Keywords*: C contains 32 keywords, and C++ supports 52 keywords.

11. *Namespace feature*: A namespace is a feature that groups the entities like classes, objects, and functions under some specific name. C does not contain the namespace feature, while C++ supports the namespace feature that avoids the name collisions.

12. *Exception handling*: C does not provide direct support to the exception handling; it needs to use functions that support exception handling. C++ provides direct support to exception handling by using a `try-catch` block.

13. *Input/Output functions*: In C, `scanf` and `printf` functions are used for input and output operations, respectively, while in C++, `cin` and `cout` are used for input and output operations, respectively.

14. *Memory allocation and de-allocation*: C supports `calloc()`

and `malloc()` functions for the memory allocation, and `free()` function for the memory de-allocation. C++ supports a new operator for the memory allocation and `delete` operator for the memory de-allocation.

15. *Inheritance*: Inheritance is a feature that allows the child class to reuse the properties of the parent class. C language does not support the inheritance while C++ supports the inheritance.

16. *Header file*: C program uses `<stdio.h>` header file while C++ program uses `<iostream.h>` header file.

1.2 C++ history

History of C++ language is interesting to know. Here we are going to discuss brief history of C++ language. C++ programming language was developed in 1980 by Bjarne Stroustrup at bell laboratories of ATT (American Telephone Telegraph), located in U.S.A. Bjarne Stroustrup is known as the founder of C++ language. It was develop for adding a feature of OOP (Object Oriented Programming) in C without significantly changing the C component. C++ programming is "relative" (called a superset) of C, it means any valid C program is also a valid C++ program. programming languages that were developed before C++ language.

Language	Year	Developed By
Algol	1960	International Group
BCPL	1967	Martin Richard
B	1970	Ken Thompson
Traditional C	1972	Dennis Ritchie
K & R C	1978	Kernighan & Dennis Ritchie

Table 1.1: Languages that were developed before C++ language

1.3 C++ Features

C++ is object oriented programming language. It provides a lot of features that are given below.

1. *Simple*: C++ is a simple language in the sense that it provides structured approach (to break the problem into parts), rich set of library functions, data types etc.

2. *Machine Independent or Portable*: Unlike assembly language, c programs can be executed in many machines with little bit or no change. But it is not platform-independent.

3. *Mid-level programming language*: C++ is also used to do low level programming. It is used to develop system applications such as kernel, driver etc. It also supports the feature of high level language. That is why it is known as mid-level language.

4. *Structured programming language*: C++ is a structured programming language in the sense that we can break the program into parts using functions. So, it is easy to understand and modify.

5. *Rich Library*: C++ provides a lot of inbuilt functions that makes the development fast.

6. *Memory Management*: It supports the feature of dynamic memory allocation. In C++ language, we can free the allocated memory at any time by calling the free() function.

7. *Speed*: The compilation and execution time of C++ language is fast.

8. *Pointer*: C++ provides the feature of pointers. We can directly interact with the memory by using the pointers. We can use pointers for memory, structures, functions, array etc.

9. *Recursion*: In C++, we can call the function within the function. It provides code reusability for every function.

10. *Extensible*: C++ language is extensible because it can easily adopt new features.

11. *Object Oriented*: C++ is object oriented programming language. OOPs makes development and maintenance easier where as in Procedure-oriented programming language it is not easy to manage if code grows as project size grows.

12. *Compiler based*: C++ is a compiler based programming language, it means without compilation no C++ program can be executed. First we need to compile our program using compiler and then we can execute our program.

1.4 Installation

There are number of ways to install and work with C++. Following are few ways:

1. Using CLI. [1]
2. POSIX-like run-time environments such as CygWin & MinGW. [2] [3]
3. Third party Integrated Development Environments (IDEs). Example: Dev-C++ [4]
4. Install C and C++ support in *Visual Studio*. [5]
5. C/C++ for *Visual Studio Code*. [6]
6. C++ on Eclipse in Windows. [7]

I suggest and will be using the last method among all the above methods for it is consistent for many reasons. One of the potential reasons is that Eclipse is an editor for many languages and also support few platforms such as mobile, desktop, web, IoT, etc. So using Eclipse always offers rather more mileage over other methods and editors.

1.4.1 Eclipse for C++

Eclipse is an integrated development environment (IDE) used in computer programming. It contains a base workspace and an extensible plug-in system for customizing the environment. Eclipse is written mostly in Java and its primary use is for developing Java applications, but it may also be used to develop applications in other programming languages via plug-ins, including Ada, ABAP, C, C++, C#, Clojure, COBOL, D, Erlang, Fortran, Groovy, Haskell, JavaScript, Julia, Lasso, Lua, Perl, PHP, Prolog, Python, R, Ruby (including Ruby on Rails framework), Rust, Scala, and Scheme.

It can also be used to develop documents with LaTeX and packages for the software Mathematica. Development environments include the Eclipse Java development tools (JDT) for Java and Scala, Eclipse CDT for C/C++, and Eclipse PDT for PHP, among others.

Install MinGW GCC or Cygwin GCC

To use Eclipse for C/C++ programming, you need a C/C++ compiler. On Windows, you could install either MinGW GCC or Cygwin GCC. Choose MinGW if you are not sure, because MinGW is lighter and easier to install, but has fewer features.

To install MinGW, go to the MinGW homepage, www.H.Bmingw.H.B

orgH.B, and follow the link to the MinGW download page. Down-

load the latest version of the MinGW installation program which should be named *MinGW<version>.exe*. While installing MinGW, at a minimum, you must install *gcc-core, gcc-g++, binutils*, and the *MinGW runtime*, but you may wish to install more. *Add the bin subdirectory of your MinGW installation to your PATH environment variable so that you can specify these tools on the command line by their simple names.*

Install Eclipse C/C++ Development Tool (CDT)

There are two ways to install CDT, depending on whether you have previously installed an Eclipse. If you have already installed "Eclipse for Java Developers" or other Eclipse packages, you could install the CDT plug-in as follows.

1. Launch Eclipse -> Help -> Install New Software -> In "Work with" field, pull down the drop-down menu and select "Kepler - http://download.eclipse.org/releases/kepler" (or juno for Eclipse 4.2; or helios for Eclipse 3.7).
2. In "Name" box, expand "Programming Language" node -> Check "C/C++ Development Tools" -> "Next" -> ... -> "Finish".

If you have not install any Eclipse package, you could download "Eclipse IDE for C/C++ Developers" from http://H.Bwww.H.B

eclipse.H.Borg/H.BdownloadsH.B, and unzip the downloaded

file into a directory of your choice.

1.4.2 Creating C++ project

Eclipse has perspectives, this means Eclipse base IDE can be made convenient for a given run-time environment. If you have installed Eclipse CDT (C++ Development Tooling), then you probably don't need to switch between perspectives. If you working with multiple run-times then probably you need to switch between perspectives in Eclipse. There are many ways to switch perspectives in Eclipse.

1. Windows short-cut key combination is *Ctl+F8*.
2. There is perspectives button at right upper corner of the Eclipse toolbar.
3. Using Eclipse main menu: Window -> Perspectives -> Open Perspective -> Other: C++

Once, you are in CDT perspective then go to File -> Create Project. After finishing necessary formalities like naming folders/files and selecting proper locations for these folders and files, you may find a folder created at left file explorers windows inside Eclipse. Right click on the main (or project home) directory, select New -> "Source File". **Try to save the file with *.cpp* format.** Now you are literally ready for C++ development. [8]

1.5 C++ Program from terminal

First let us understand as how to compile and run C++ files from terminal. Create a nice place, you may call it as workspace, as every programming language need to have certain workspace and namespace for running their code. At times, it can be a simple

folder in which we are going to organize our scripts. In my computer I created one such folder and it looks like the one below:

```
D:\Work\Books\CPP\scripts>dir
 Volume in drive D is Data
 Volume Serial Number is 0C88-924D

 Directory of D:\Work\Books\CPP\scripts

24-10-2021  18:20    <DIR>          .
24-10-2021  18:20    <DIR>          ..
24-10-2021  18:19                 0 demo.cpp
               1 File(s)              0 bytes
               2 Dir(s)  659,893,870,592 bytes free

D:\Work\Books\CPP\scripts>
```

`D:` is the workspace for my C++ programs. If you see I have one file with name `demo.cpp`. This is the file in which I am going to write C++ program. Let's try the following code snippet in plain text file in Windows known as *Notepad*.

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

int main(){
cout << "Hello World!" << endl;
cout << "This is first C++ program.";
return 0;
}
```

Now in the terminal use the following statement to compile this program.

```
g++ demo.cpp
```

This shall produce the following changes in the same directory.

```
D:\Work\Books\CPP\scripts>dir
 Volume in drive D is Data
 Volume Serial Number is 0C88-924D
```

```
 Directory of D:\Work\Books\CPP\scripts

24-10-2021  18:25    <DIR>          .
24-10-2021  18:25    <DIR>          ..
24-10-2021  18:25            56,932 a.exe
24-10-2021  18:24               147 demo.cpp
               2 File(s)         57,079 bytes
               2 Dir(s)  659,893,624,832 bytes free
```

A new file *a.exe* is produced which is an Windows executable file. Now if you run `a.exe` in the terminal it shall produce the output.

```
D:\Work\Books\CPP\scripts>a.exe
Hello World!
This is first C++ program.
```

As far as compiling is concerned, there are number of ways to compile a C++ program into variety of output formats.

1. `g++ -S`: generates a *file_name.s* known as assembly source file.
2. `g++ -c`: generates a *file_name.o* object code file in present working directory.
3. `g++ -o/exe`: generates executable target file with `target_name` (or a.out by default).

Suppose I want to compile to an output file with a given name. I need to execute the same statement with few changes shown as below.

```
g++ -o main.exe demo.cpp
```

This will create `main.exe` in the same filder/directory.

1.5.1 Object code

Object code is highly useful to *compile and link multiple files*. I want to plan my project as below:

1. Create a header (.h) file with method definitions (`helloworld.h`).
2. Define the method (`helloworld.cpp`).

3. Invoke the method from *main* file (`hello.cpp`).

So I am planning my project with one header file and two *cpp* files. Keep all the files in the same directory. The code inside these files is going to be as below:

```
// hello.cpp file
#include "helloWorld.h"
#include <iostream>
int main()
{
    std::cout << "This is \n";
    helloWorld();
std::cout << "greeting";
    return 0;
}

// helloWorld.cpp file
#include <iostream>
void helloWorld()
{
    std::cout << "Hello World\n";
}

// helloWorld.h file
void helloWorld();
```

Now run the following statements in the terminal.

```
g++ -c helloWorld.cpp hello.cpp
g++ -o main.exe hello.o helloworld.o
```

This shall produce executable file with name `main.exe`, which can produce the below output.

```
This is
Hello World
greeting
```

1.6 Eclipse for C++ development

In the main menu go to File -> New and create Source File, but save the file with *.cpp* extension. This can also be done by right clicking project directory. Eclipse will display a file which is just a normal text file but with *.cpp* extension. Write the following code in it.

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

int main(){
cout << "Hello World!" << endl;
cout << "This is first C++ program.";
return 0;
}
```

Go to Project -> Build All (Ctl + B). This action shall produce a couple of additional folders and files in those folder. Now go to Run -> Run As -> Local C/C++ Application (Ctl + F11). This shall produce the following output in the Eclipse Console.

```
Hello World!
This is first C++ program.
```

1.7 Basic Input/Output

C++ I/O operation is using the stream concept. Stream is the sequence of bytes or flow of data. It makes the performance fast. If bytes flow from main memory to device like printer, display screen, or a network connection, etc, this is called as output operation. If bytes flow from device like printer, display screen, or a network connection, etc to main memory, this is called as input operation.

1.7.1 I/O Library Header Files

Let us see the common header files used in C++ programming are:

1. `include<stdio.h>`: It is used to perform input and output operations using functions `scanf()` and `printf()`.

2. `include<iostream>`: It is used as a stream of Input and Output using `cin` and `cout`.
3. `include<string.h>`: It is used to perform various features related to string manipulation like `strlen(), strcmp(), strcpy(), size()`, etc.
4. `include<math.h>`: It is used to perform mathematical operations like `sqrt(), log2(), pow()`, etc.
5. `include<iomanip.h>`: It is used to access `setw()` and `setprecision()` function to limit the decimal places in variables.
6. `include<signal.h>`: It is used to perform signal handling functions like `signal()` and `raise()`.
7. `include<stdarg.h>`:It is used to perform standard argument functions like `va_start()` and `va_arg()`. It is also used to indicate start of the variable-length argument list and to fetch the arguments from the variable-length argument list in the program respectively.
8. `include<errno.h>`: It is used to perform error handling operations like `errno(), strerror(), perror()`, etc.
9. `include<fstream.h>`: It is used to control the data to read from a file as an input and data to write into the file as an output.
10. `include<time.h>`: It is used to perform functions related to `date()` and `time()` like `setdate()` and `getdate()`. It is also used to modify the system date and get the CPU time respectively.
11. `include<float.h>`: It contains a set of various platform-dependent constants related to floating point values. These constants are proposed by ANSI C. They allow making programs more portable. Some examples of constants included in this header file are- e(exponent), b(base/radix), etc.
12. `include<limits.h>`: It determines various properties of the various variable types. The macros defined in this header, limits the values of various variable types like char, int, and long. These limits specify that a variable cannot store any

value beyond these limits, for example an unsigned character can store up to a maximum value of 255.

1.7.2 Standard output stream (cout)

The cout is a predefined object of ostream class. It is connected with the standard output device, which is usually a display screen. The cout is used in conjunction with stream insertion operator (<<) to display the output on a console. Let's see the simple example of standard output stream (cout):

```
#include <iostream>
using namespace std;
int main( ) {
   char ary[] = "Welcome to C++ tutorial";
   cout << "Value of ary is: " << ary << endl;
}
```

Output:

```
Value of ary is: Welcome to C++ tutorial
```

1.7.3 Standard input stream (cin)

The cin is a predefined object of istream class. It is connected with the standard input device, which is usually a keyboard. The cin is used in conjunction with stream extraction operator (>>) to read the input from a console. Let's see the simple example of standard input stream (cin):

```
#include <iostream>
using namespace std;
int main( ) {
  int age;
   cout << "Enter your age: ";
   cin >> age;
   cout << "Your age is: " << age << endl;
}
```

Output:

```
Enter your age: 22
```

```
Your age is: 22
```

1.7.4 Standard end line (endl)

The `endl` is a predefined object of `ostream` class. It is used to insert a new line characters and flushes the stream. Let's see the simple example of standard end line (`endl`):

```
#include <iostream>
using namespace std;
int main( ) {
cout << "C++ Tutorial";
cout << " Javatpoint"<<endl;
cout << "End of line"<<endl;
}
```

Output:

```
C++ Tutorial Javatpoint
End of line
```

1.8 Identifiers and Keywords

1.8.1 Identifiers

C++ identifiers in a program are used to refer to the name of the *variables, functions, arrays*, or other *user-defined data types* created by the programmer. They are the basic requirement of any language. Every language has its own rules for naming the identifiers. In short, we can say that the C++ identifiers represent the essential elements in a program which are given below:

1. Constants
2. Variables
3. Functions
4. Labels
5. Defined data types

1.8.2 Naming rules

1. Only alphabetic characters, digits, and underscores are allowed.
2. The identifier name cannot start with a digit, i.e., the first letter should be alphabetical. After the first letter, we can use letters, digits, or underscores.
3. In C++, uppercase and lowercase letters are distinct. Therefore, we can say that C++ identifiers are case-sensitive.
4. A declared keyword cannot be used as a variable name.

For example, suppose we have two identifiers, named as 'FirstName', and 'Firstname'. Both the identifiers will be different as the letter 'N' in the first case in uppercase while lowercase in second. Therefore, it proves that identifiers are case-sensitive. Following are few valid Identifiers

1. Result
2. Test2
3. _sum
4. power

The following are the examples of invalid identifiers:

1. Sum-1 // containing special character '-'.
2. 2data // the first letter is a digit.
3. break // use of a keyword.

The major difference between C and C++ is the limit on the length of the name of the variable. ANSI C considers only the first 32 characters in a name while ANSI C++ imposes no limit on the length of the name. Constants are the identifiers that refer to the fixed value, which do not change during the execution of a program. Both C and C++ support various kinds of literal constants, and they do have any memory location. For example, 123, 12.34, 037, 0X2, etc. are the literal constants. Let's look at a simple example to understand the concept of identifiers.

```
#include <iostream>
using namespace std;
int main()
{
  int a;
  int A;
  cout<<"Enter the values of 'a' and 'A'";
  cin>>a;
  cin>>A;
  cout<<"\nThe values that you have entered are : "<<a<<" ,
      ↪ "<<A;
  return 0;
}
```

In the above code, we declare two variables 'a' and 'A'. Both the letters are same but they will behave as different identifiers. As we know that the identifiers are the case-sensitive so both the identifiers will have different memory locations.

1.8.3 Keywords

Keywords are the reserved words that have a special meaning to the compiler. They are reserved for a special purpose, which cannot be used as the identifiers. For example, 'for', 'break', 'while', 'if', 'else', etc. are the predefined words where predefined words are those words whose meaning is already known by the compiler. Whereas, the identifiers are the names which are defined by the programmer to the program elements such as variables, functions, arrays, objects, classes.

Table 1.2 has the list of all C++ keywords. (as of C++17)

The Table 1.3 has the list of differences between identifiers and keywords:

1.9 Data Types

All variables use data-type during declaration to restrict the type of data to be stored. Therefore, we can say that data types are used to tell the variables the type of data it can store. Whenever a variable is defined in C++, the compiler allocates some memory

alignas	decltype	namespace	struct
alignof	default	new	switch
and	delete	noexcept	template
and_eq	do	not	this
asm	double	not_eq	thread_local
auto	dynamic_cast	nullptr	throw
bitand	else	operator	true
bitor	enum	or	try
bool	explicit	or_eq	typedef
break	export	private	typeid
case	extern	protected	typename
catch	false	public	union
char	float	register	unsigned
char16_t	for	reinterpret_cast	using
char32_t	friend	return	virtual
class	goto	short	void
compl	if	signed	volatile
const	inline	sizeof	wchar_t
constexpr	int	static	while
const_cast	long	static_assert	xor
continue	mutable	static_cast	xor_eq

Table 1.2: Keywords in C++

for that variable based on the data-type with which it is declared. Every data type requires a different amount of memory.

A data type specifies the type of data that a variable can store such as integer, floating, character etc. There are 4 types of data types in C++ language. Data types in C++ is mainly divided into three types:

1. **Primitive Data Types:** These data types are built-in or predefined data types and can be used directly by the user to declare variables. example: int, char , float, bool etc. Primitive data types available in C++ are:

    - Integer
    - Character

Identifiers	Keywords
Identifiers are the names defined by the programmer to the basic elements of a program.	Keywords are the reserved words whose meaning is known by the compiler.
It is used to identify the name of the variable.	It is used to specify the type of entity.
It can consist of letters, digits, and underscore.	It contains only letters.
It can use both lowercase and uppercase letters.	It uses only lowercase letters.
No special character can be used except the underscore.	It cannot contain any special character.
The starting letter of identifiers can be lowercase, uppercase or underscore.	It can be started only with the lowercase letter.
It can be classified as internal and external identifiers.	It cannot be further classified.
Examples are test, result, sum, power, etc.	Examples are 'for', 'if', 'else', 'break', etc.

Table 1.3: Differences between identifiers and keywords.

- Boolean
- Floating Point
- Double Floating Point
- Valueless or Void
- Wide Character

2. **Derived Data Types:** The data-types that are derived from the primitive or built-in datatypes are referred to as Derived Data Types. These can be of four types namely:
   - Function
   - Array
   - Pointer

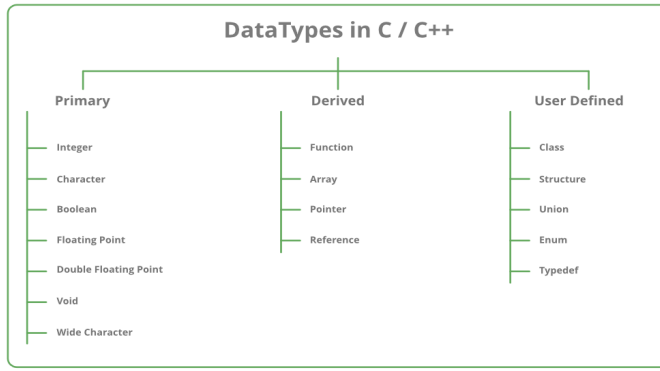


Figure 1.1: Data types in C

- Reference

3. **Abstract or User-Defined Data Types:** These data types are defined by user itself. Like, defining a class in C++ or a structure. C++ provides the following user-defined datatypes:
   - Class
   - Structure
   - Union
   - Enumeration
   - Typedef defined DataType

1.9.1 Basic Data Types

The basic data types are integer-based and floating-point based. C++ language supports both signed and unsigned literals. The memory size of basic data types may change according to 32 or 64 bit operating system. Let's see the basic data types. It size is given according to 32 bit OS.

1. **Integer:** Keyword used for integer data types is int. Integers typically requires 4 bytes of memory space and ranges from -2147483648 to 2147483647.

2. **Character**: Character data type is used for storing characters. Keyword used for character data type is char. Characters typically requires 1 byte of memory space and ranges from -128 to 127 or 0 to 255.

3. **Boolean**: Boolean data type is used for storing boolean or logical values. A boolean variable can store either true or false. Keyword used for boolean data type is bool.

4. **Floating Point**: Floating Point data type is used for storing single precision floating point values or decimal values. Keyword used for floating point data type is float. Float variables typically requires 4 byte of memory space.

5. **Double Floating Point:** Double Floating Point data type is used for storing double precision floating point values or decimal values. Keyword used for double floating point data type is double. Double variables typically requires 8 byte of memory space.

6. **void:** Void means without any value. void datatype represents a valueless entity. Void data type is used for those function which does not returns a value.

7. **Wide Character**: Wide character data type is also a character data type but this data type has size greater than the normal 8-bit datatype. Represented by wchar_t. It is generally 2 or 4 bytes long.

The Table 1.4 shows the tabular representation for above description.

1.9.2 Datatype Modifiers

As the name implies, data type modifiers are used with the built-in data types to modify the length of data that a particular data type can hold.

Data type modifiers available in C++ are:

1. Signed

2. Unsigned

Data Types	Memory Size	Range
char	1 byte	-128 to 127
signed char	1 byte	-128 to 127
unsigned char	1 byte	0 to 127
short	2 byte	-32,768 to 32,767
signed short	2 byte	-32,768 to 32,767
unsigned short	2 byte	0 to 32,767
int	2 byte	-32,768 to 32,767
signed int	2 byte	-32,768 to 32,767
unsigned int	2 byte	0 to 32,767
short int	2 byte	-32,768 to 32,767
signed short int	2 byte	-32,768 to 32,767
unsigned short int	2 byte	0 to 32,767
long int	4 byte	
signed long int	4 byte	
unsigned long int	4 byte	
float	4 byte	
double	8 byte	
long double	10 byte	

Table 1.4: Basic data types

3. Short
4. Long

Table 1.4 summarizes the modified size and range of built-in data types when combined with the type modifiers. These values may vary from compiler to compiler. Values in the above example corresponds to GCC 32 bit. However, it is possible to display the size of all the data types by using the `sizeof()` operator and passing the keyword of the data type as argument to this function as shown below:

```
// C++ program to sizes of data types
#include<iostream>
using namespace std;

int main()
{
  cout << "Size of char : " << sizeof(char)
    << " byte" << endl;
```

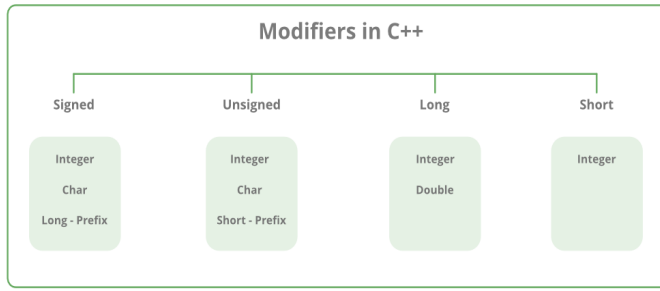


Figure 1.2: Modifiers

```
  cout << "Size of int : " << sizeof(int)
    << " bytes" << endl;
  cout << "Size of short int : " << sizeof(short int)
    << " bytes" << endl;
  cout << "Size of long int : " << sizeof(long int)
     << " bytes" << endl;
  cout << "Size of signed long int : " << sizeof(signed long
      ↪ int)
     << " bytes" << endl;
  cout << "Size of unsigned long int : " << sizeof(unsigned
      ↪ long int)
     << " bytes" << endl;
  cout << "Size of float : " << sizeof(float)
     << " bytes" <<endl;
  cout << "Size of double : " << sizeof(double)
     << " bytes" << endl;
  cout << "Size of wchar_t : " << sizeof(wchar_t)
     << " bytes" <<endl;

  return 0;
}
```

Output:

```
Size of char : 1 byte
Size of int : 4 bytes
Size of short int : 2 bytes
Size of long int : 8 bytes
Size of signed long int : 8 bytes
Size of unsigned long int : 8 bytes
Size of float : 4 bytes
Size of double : 8 bytes
Size of wchar\_t : 4 bytes
```

1.10 C++ Variables, Literals and Constants

1.10.1 C++ Variables

In programming, a variable is a container (storage area) to hold data. To indicate the storage area, each variable should be given a unique name (identifier). For example,

```
int age = 14;
```

Here, `age` is a variable of the `int` data type, and we have assigned an integer value 14 to it. he `int` data type suggests that the variable can only hold integers. Similarly, we can use the `double` data type if we have to store decimals and exponential. The value of a variable can be changed, hence the name variable.

```
int age = 14;   // age is 14
age = 17;       // age is 17
```

1.10.2 Rules for naming a variable

1. A variable name can only have alphabets, numbers, and the underscore _.
2. A variable name cannot begin with a number.
3. Variable names should not begin with an uppercase character.
4. A variable name cannot be a keyword. For example, int is a keyword that is used to denote integers.
5. A variable name can start with an underscore. However, it's not considered a good practice.

1.10.3 C++ Literals

Literals are data used for representing *fixed values*. They can be used directly in the code. For example: 1, 2.5, 'c' etc. Here, 1, 2.5 and 'c' are literals. Why? You cannot assign different values to these terms. Here's a list of different literals in C++ programming.

1. **Integers:**
   An integer is a numeric literal(associated with numbers) without any fractional or exponential part. There are three types of integer literals in C programming:

   - decimal (base 10)
   - octal (base 8)
   - hexadecimal (base 16)

   In C++ programming, octal starts with a 0, and hexadecimal starts with a 0x.

2. **Floating-point Literals:**
   A floating-point literal is a numeric literal that has either a fractional form or an exponent form. For example: -2.0; 0.0000234; -0.22E-5 (E-5 = 10-5)

3. **Characters:**
   A character literal is created by enclosing a single character inside single quotation marks. For example: 'a', 'm', 'F', '2', '}' etc.

4. **Escape Sequences:** Sometimes, it is necessary to use characters that cannot be typed or has special meaning in C++ programming. For example, newline (enter), tab, question mark, etc. In order to use these characters, escape sequences are used.

5. **String Literals:** A string literal is a sequence of characters enclosed in double-quote marks. For example:

1.10.4 C++ Constants

In C++, we can create variables whose value cannot be changed. For that, we use the `const` keyword. Here's an example:

```
const int LIGHT\_SPEED = 299792458;
LIGHT\_SPEED = 2500 // Error! LIGHT_SPEED is a constant.
```

Here, we have used the keyword `const` to declare a constant named $LIGHT\_SPEED$. If we try to change the value of $LIGHT\_SPEED$, we will get an error.

Escape Sequences	Characters
\b	Backspace
\f	Form feed
\n	Newline
\r	Return
\t	Horizontal tab
\v	Vertical tab
\$	Backslash
\'	Single quotation mark
\"	Double quotation mark
\?	Question mark
\0	Null Character

Table 1.5: Escape characters

"good"	string constant
""	null string constant
" "	string constant of six white space
"x"	string constant having a single character
"Earth is round"	prints string with a newline

Table 1.6: String literals

1.11 Operators and Expressions

C++ expression consists of operators, constants, and variables which are arranged according to the rules of the language. It can also contain function calls which return values. Operators are the foundation of any programming language. Thus the functionality of the C/C++ programming language is incomplete without the use of operators. We can define operators as symbols that help us to perform specific mathematical and logical computations on operands. In other words, we can say that an operator operates the operands. An expression can consist of one or more operands, zero or more operators to compute a value. Every expression produces some value which is assigned to the variable with the help of an assignment operator.

1.11.1 Operators

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. C++ is rich in built-in operators and provide the following types of operators.

1. Arithmetic Operators
2. Relational Operators
3. Logical Operators
4. Bitwise Operators
5. Assignment Operators
6. Misc Operators

This chapter will examine the arithmetic, relational, logical, bitwise, assignment and other operators one by one.

Arithmetic Operators

There are following arithmetic operators (Table 1.7) supported by C++ language. Assume variable A holds 10 and variable B holds 20, then

Operator	Description	Example
+	Adds two operands	A + B will give 30
-	Subtracts second operand from the first	A - B will give -10
*	Multiplies both operands	A * B will give 200
	Divides numerator by de-numerator	B / A will give 2
%	Modulus Operator and remainder of after an	B % A will give 0
++	Increment operator, increases integer division integer value by one	A++ will give 11
−−	Decrement operator, decreases integer value by one.	A−− will give 9

Table 1.7:

Relational Operators

There are following (Table 1.8) relational operators supported by C++ language. Assume variable A holds 10 and variable B holds 20, then

Operator	Description	Example
==	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A == B) is not true.
! =	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A ! = B) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

Table 1.8: Relational operators

Logical operators

There are following logical operators (Table 1.9) supported by C++ language. Assume variable A holds 1 and variable B holds 0, then

Operator	Description	Example
&&	Called Logical AND operator. If both the operands are non-zero, then condition becomes true.	(A && B) is false.
||	Called Logical OR Operator. If any of the two operands is non-zero, then condition becomes true.	(A || B) is true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true, then Logical NOT operator will make false.	!(A&&B) is true.

Table 1.9: Logical operators

Bitwise Operators

Bitwise operator works on bits and perform bit-by-bit operation. The truth tables for &, ||, and ^ are as follows (Table 1.10)

Assume if A = 60; and B = 13; now in binary format they will be as follows

A = 0011 1100 B = 0000 1101 ———— $A\&B$ = 0000 1100
$A\|B$ = 0011 1101 AB = 0011 0001 = 1100 0011

p	q	$p\&q$	$p\|\|q$	pq
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

Table 1.10: Bitwise operators

The Bitwise operators supported by C++ language are listed in the following table 1.11. Assume variable A holds 60 and variable B holds 13, then

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12 which is 0000 1100
|	Binary OR Operator copies a bit if it exists in either operand.	(A | B) will give 61 which is 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) will give 49 which is 0011 0001
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) will give -61 which is 1100 0011 in 2's complement form due to a signed binary number.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A « 2 will give 240 which is 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A » 2 will give 15 which is 0000 1111

Table 1.11: Bitwise operators

Assignment Operators

There are following assignment operators supported by C++ language (Table 1.12).

Misc Operators

The following table lists some other operators that C++ supports.

1. `sizeof`
   sizeof operator returns the size of a variable. For example, `sizeof(a)`, where 'a' is integer, and will return 4.

Operator	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand.	C = A + B will assign value of A + B into C.
+ =	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand.	C += A is equivalent to C = C + A
− =	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand.	C -= A is equivalent to C = C - A
* =	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand.	C *= A is equivalent to C = C * A
/ =	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand.	C /= A is equivalent to C = C / A
% =	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand.	C % = A is equivalent to C = C % A
<<=	Left shift AND assignment operator.	C «= 2 is same as C = C « 2
>>=	Right shift AND assignment operator.	C »= 2 is same as C = C » 2
& =	Bitwise AND assignment operator.	C & = 2 is same as C = C & 2
^=	Bitwise exclusive OR and assignment operator.	C ^= 2 is same as C = C ^ 2
| =	Bitwise inclusive OR and assignment operator.	C |= 2 is same as C = C | 2

Table 1.12: Assignment operator

2. `Condition ? X : Y`
   Conditional operator (?). If Condition is true then it returns value of X otherwise returns value of Y.

3. `,`
   Comma operator causes a sequence of operations to be performed. The value of the entire comma expression is the value of the last expression of the comma-separated list.

4. . (dot) and -> (arrow)
   Member operators are used to reference individual members of classes, structures, and unions.

5. `Cast`
   Casting operators convert one data type to another. For example, `int(2.2000)` would return 2.

6. `&`
   Pointer operator & returns the address of a variable. For example &*a*; will give actual address of the variable.

7. *
   Pointer operator * is pointer to a variable. For example `*var`; will pointer to a variable var.

Operators Precedence in C++

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator. For example x = 7 + 3 * 2; here, x is assigned 13, not 20 because operator * has higher precedence than +, so it first gets multiplied with 3*2 and then adds into 7. Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

Category	Operator	Associativity
Postfix	()[]− > . + + − −	Left to right
Unary	+−˜+ + − − (type)* & sizeof	Right to left
Multiplicative	& ∗%	Left to right
Additive	+−	Left to right
Shift	<<>>	Left to right
Relational	<<=>>=	Left to right
Equality	==! =	Left to right
Bitwise AND	&	Left to right
Bitwise XOR		Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	? :	Right to left
Assignment	= + = − = ∗ = / = % =>>=<<= & =ˆ= | =	Right to left
Comma	,	Left to right

Table 1.13: Operator precedence

1.11.2 C++ Expression

An expression is a combination of operators, constants and variables. An expression may consist of one or more operands, and zero or more operators to produce a value.

An expression can be of following types

1. Constant expressions
2. Integral expressions

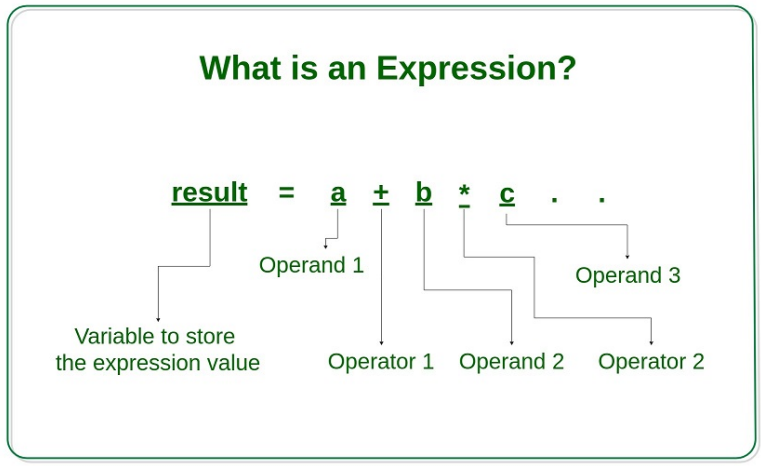


Figure 1.3:

3. Float expressions
4. Pointer expressions
5. Relational expressions
6. Logical expressions
7. Bitwise expressions
8. Special assignment expressions

If the expression is a combination of the above expressions, such expressions are known as *compound expressions*.

Constant expressions

A constant expression is an expression that consists of only constant values. It is an expression whose value is determined at the compile-time but evaluated at the run-time. It can be composed of integer, character, floating-point, and enumeration constants. Constants are used in the following situations:

1. It is used in the *subscript declarator* to describe the array bound.
2. It is used after the case keyword in the switch statement.
3. It is used as a numeric value in an enum
4. It specifies a bit-field width.
5. It is used in the pre-processor `if`
6. In the above scenarios, the constant expression can have integer, character, and enumeration constants. We can use the
7. static and `extern` keyword with the constants to define the function-scope.

The following table shows the expression containing constant value:

Expression containing constant	Constant value
$x = (2/3) * 4$	$(2/3) * 4$
$externinty = 67$	67
$intz = 43$	43
$staticinta = 56$	56

Table 1.14: Expressions

Let's see a simple program containing constant expression:

```
#include <iostream>
using namespace std;
int main()
{
  int x;        // variable declaration.
  x=(3/2) + 2;  // constant expression
  cout<<"Value of x is : "<<x;  // displaying the value of x
      ↪ .
  return 0;
}
```

In the above code, we have first declared the 'x' variable of integer type. After declaration, we assign the simple constant expression to the 'x' variable.

Output

Value of x is : 3

Integral Expressions

An integer expression is an expression that produces the integer value as output after performing all the explicit and implicit conversions. Following are the examples of integral expression:

```
(x * y) -5
x + int(9.0)
where x and y are the integers.
```

Let's see a simple example of integral expression:

```
#include <iostream>
using namespace std;
int main()
{
  int x;  // variable declaration.
  int y;  // variable declaration
  int z;  // variable declaration
  cout<<"Enter the values of x and y";
  cin>>x>>y;
  z=x+y;
  cout<<"\n"<<"Value of z is :"<<z; //  displaying the value
      ↪  of z.
  return 0;
}
```

In the above code, we have declared three variables, i.e., x, y, and z. After declaration, we take the user input for the values of 'x' and 'y'. Then, we add the values of 'x' and 'y' and stores their result in 'z' variable.

Output

Enter the values of x and y 8 9 Value of z is :17

Let's see another example of integral expression.

```
#include <iostream>
using namespace std;
int main()
{
   int x;   // variable declaration
   int y=9;    // variable initialization
   x=y+int(10.0);    // integral expression
  cout<<"Value of x : "<<x;   // displaying the value of x.
  return 0;
}
```

In the above code, we declare two variables, i.e., x and y. We store the value of expression `(y+int(10.0))` in a 'x' variable.

Output

```
Value of x : 19
```

Float Expressions

A float expression is an expression that produces floating-point value as output after performing all the explicit and implicit conversions. The following are the examples of float expressions:

```
x+y
(x/10) + y
34.5
x+float(10)
```

Let's understand through an example.

```
#include <iostream>
using namespace std;
int main()
{
   float x=8.9;      // variable initialization
   float y=5.6;      // variable initialization
   float z;             // variable declaration
   z=x+y;
   std::cout <<"value of z is :"  << z<<std::endl;  //
       ↪ displaying the value of z.
    return 0;
}
```

Output

value of z is :14.5

Let's see another example of float expression.

```
#include <iostream>
using namespace std;
int main()
{
   float x=6.7;    // variable initialization
   float y;      // variable declaration
   y=x+float(10);   // float expression
   std::cout <<"value of y is :"  << y<<std::endl;  //
       ↪ displaying the value of y
```

```
    return 0;
}
```

In the above code, we have declared two variables, i.e., x and y. After declaration, we store the value of expression `(x+float(10))` in variable 'y'.

Output

value of y is :16.7

Pointer Expressions

A pointer expression is an expression that produces address value as an output. The following are the examples of pointer expression:

x ptr ptr++ ptr-

Let's understand through an example.

```
#include <iostream>
using namespace std;
int main()
{
   int a[]={1,2,3,4,5};  // array initialization
   int *ptr;        // pointer declaration
   ptr=a;    // assigning base address of array to the
       ↪ pointer ptr
   ptr=ptr+1;   // incrementing the value of pointer
   std::cout <<"value of second element of an array : "  <<
       ↪ *ptr<<std::endl;
   return 0;
}
```

In the above code, we declare the array and a pointer ptr. We assign the base address to the variable 'ptr'. After assigning the address, we increment the value of pointer 'ptr'. When pointer is incremented then 'ptr' will be pointing to the second element of the array.

Output

```
value of second element of an array : 2
```

Relational Expressions

A relational expression is an expression that produces a value of type bool, which can be either true or false. It is also known as a boolean expression. When arithmetic expressions are used on both sides of the relational operator, arithmetic expressions are evaluated first, and then their results are compared. The following are the examples of the relational expression:

```
a>b
a-b >= x-y
a+b>80
```

Let's understand through an example

```
#include <iostream>
using namespace std;
int main()
{
  int a=45;    // variable declaration
  int b=78;    // variable declaration
  bool y= a>b;   // relational expression
  cout<<"Value of y is :"<<y;  // displaying the value of y.
  return 0;
}
```

In the above code, we have declared two variables, i.e., 'a' and 'b'. After declaration, we have applied the relational operator between the variables to check whether 'a' is greater than 'b' or not.

Output

Value of y is :0

Let's see another example.

```
#include <iostream>
using namespace std;
int main()
{
 int a=4;     // variable declaration
 int b=5;     // variable declaration
 int x=3;     // variable declaration
 int y=6;    // variable declaration
 cout<<((a+b)>=(x+y));   // relational expression
 return 0;
}
```

In the above code, we have declared four variables, i.e., 'a', 'b', 'x' and 'y'. Then, we apply the relational operator (>=) between these variables.

Output

1

Logical Expressions

A logical expression is an expression that combines two or more relational expressions and produces a bool type value. The logical operators are '&&' and '||' that combines two or more relational expressions. The following are some examples of logical expressions:

a>b x>y a>10 || b==5

Let's see a simple example of logical expression.

```
#include <iostream>
using namespace std;
int main()
{
 int a=2;
 int b=7;
 int c=4;
cout<<((a>b)||(a>c));
return 0;
}
```

Output

0

Bitwise Expressions

A bit-wise expression is an expression which is used to manipulate the data at a bit level. They are basically used to shift the bits. For example:

```
x=3
x>>3 // This statement means that we are shifting the three-
    ↪ bit position to the right.
```

In the above example, the value of 'x' is 3 and its binary value is 0011. We are shifting the value of 'x' by three-bit position to the

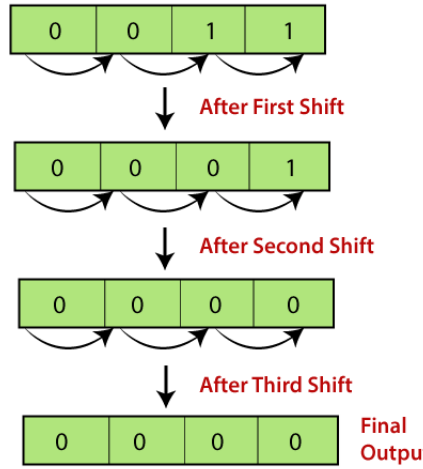


Figure 1.4: Bitwise shift

right. Let's understand through the diagrammatic representation. Let's see a simple example.

```
#include <iostream>
using namespace std;
int main()
{
  int x=5;   // variable declaration
  std::cout << (x>>1) << std::endl;
  return 0;
}
```

In the above code, we have declared a variable 'x'. After declaration, we applied the bitwise operator, i.e., right shift operator to shift one-bit position to right.

Output

2

Let's look at another example.

```
#include <iostream>
```

```
using namespace std;
int main()
{
 int x=7;   // variable declaration
std::cout << (x<<3) << std::endl;
return 0;
}
```

In the above code, we have declared a variable 'x'. After declaration, we applied the left shift operator to variable 'x' to shift the three-bit position to the left.

Output

56

Special Assignment Expressions

Special assignment expressions are the expressions which can be further classified depending upon the value assigned to the variable. Few special assignment expressions are as follows.

1. Chained Assignment
2. Embedded Assignment Expression
3. Compound Assignment

Chained assignment expression is an expression in which the same value is assigned to more than one variable by using single statement. For example:

```
a=b=20
 or
(a=b) = 20
```

Let's understand through an example.

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

 int a;   // variable declaration
 int b;   // variable declaration
 a=b=80;  // chained assignment
 std::cout <<"Values of 'a' and 'b' are : " <<a<<","<<b<<
     ↪ std::endl;
```

```
 return 0;
}
```

In the above code, we have declared two variables, i.e., 'a' and 'b'. Then, we have assigned the same value to both the variables using chained assignment expression.

Output

Values of 'a' and 'b' are : 80,80

Note: Using chained assignment expression, the value cannot be assigned to the variable at the time of declaration. For example, int a=b=c=90 is an invalid statement.

Embedded Assignment Expression

An embedded assignment expression is an assignment expression in which assignment expression is enclosed within another assignment expression. Let's understand through an example.

```
#include <iostream>
using namespace std;
int main()
{
 int a;  // variable declaration
 int b;  // variable declaration
 a=10+(b=90);  // embedded assignment expression
 std::cout <<"Values of 'a' is " <<a<< std::endl;
 return 0;
}
```

In the above code, we have declared two variables, i.e., 'a' and 'b'. Then, we applied embedded assignment expression (a=10+(b=90)).

Output

Values of 'a' is 100

Compound Assignment

A compound assignment expression is an expression which is a combination of an assignment operator and binary operator. For example,

a+=10;

In the above statement, 'a' is a variable and '+=' is a compound statement. Let's understand through an example.

```
#include <iostream>
using namespace std;
int main()
{
  int a=10;   // variable declaration
  a+=10;    // compound assignment
  std::cout << "Value of a is :" <<a<< std::endl; //
      ↪ displaying the value of a.
  return 0;
}
```

In the above code, we have declared a variable 'a' and assigns 10 value to this variable. Then, we applied compound assignment operator (+=) to 'a' variable, i.e., a+=10 which is equal to (a=a+10). This statement increments the value of 'a' by 10.

Output

Value of a is :20

Chapter 2

Functional programming

2.1 Conditional statements

In C++ programming, if statement is used to test the condition. There are various types of if statements in C++.

1. if statement
2. if-else statement
3. nested if statement
4. if-else-if ladder

2.1.1 C++ IF Statement

The C++ if statement tests the condition. It is executed if condition is true.

if(condition) //code to be executed

C++ If Example

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

```
3
4  int main () {
5   int num = 10;
6    if (num % 2 == 0)
7    {
8      cout<<"It is even number";
9    }
10   return 0;
11 }
```

Output:

It is even number

2.1.2 C++ IF-else Statement

The C++ if-else statement also tests the condition. It executes if block if condition is true otherwise else block is executed. The syntax for if-else is as follows.

```
1 if(condition){
2 //code if condition is true
3 }else{
4 //code if condition is false
5 }
```

C++ If-else Example

```
1  #include <iostream>
2  using namespace std;
3  int main () {
4     int num = 11;
5    if (num % 2 == 0)
6    {
7      cout<<"It is even number";
8    }
9    else
10   {
11     cout<<"It is odd number";
12   }
13    return 0;
14 }
```

Output:

It is odd number

C++ If-else Example: with input from user

```
#include <iostream>
using namespace std;
int main () {
  int num;
  cout<<"Enter a Number: ";
  cin>>num;
  if (num % 2 == 0)
  {
    cout<<"It is even number"<<endl;
  }
  else
  {
    cout<<"It is odd number"<<endl;
  }
  return 0;
}
```

Output:

```
Enter a number:11
It is odd number
```

Output:

```
Enter a number:12
It is even number
```

2.1.3 C++ IF-else-if ladder Statement

The C++ if-else-if ladder statement executes one condition from multiple statements. Following is the syntax for if-else-if ladder in C++.

```
if(condition1){
//code to be executed if condition1 is true
}else if(condition2){
//code to be executed if condition2 is true
}
else if(condition3){
//code to be executed if condition3 is true
}
...
else{
//code to be executed if all the conditions are false
}
```

C++ If else-if Example

```
#include <iostream>
using namespace std;
int main () {
 int num;
 cout<<"Enter a number to check grade:";
 cin>>num;
  if (num <0 || num >100)
  {
    cout<<"wrong number";
  }
  else if(num >= 0 && num < 50){
    cout<<"Fail";
  }
  else if (num >= 50 && num < 60)
  {
    cout<<"D Grade";
  }
  else if (num >= 60 && num < 70)
  {
    cout<<"C Grade";
  }
  else if (num >= 70 && num < 80)
  {
    cout<<"B Grade";
  }
  else if (num >= 80 && num < 90)
  {
    cout<<"A Grade";
  }
  else if (num >= 90 && num <= 100)
  {
    cout<<"A+ Grade";
  }
    }
```

Output:

```
Enter a number to check grade:66
C Grade

Enter a number to check grade:-2
wrong number
```

2.2 C++ Switch

The C++ switch statement executes one statement from multiple conditions. It is like if-else-if ladder statement in C++.

```
switch(expression){
case value1:
 //code to be executed;
 break;
case value2:
 //code to be executed;
 break;
......

default:
 //code to be executed if all cases are not matched;
 break;
}
```

C++ Switch Example

```
#include <iostream>
using namespace std;
int main () {
 int num;
 cout<<"Enter a number to check grade:";
 cin>>num;
   switch (num)
  {
    case 10: cout<<"It is 10"; break;
    case 20: cout<<"It is 20"; break;
    case 30: cout<<"It is 30"; break;
    default: cout<<"Not 10, 20 or 30"; break;
  }
}
```

Output:

```
Enter a number:
10
It is 10

Enter a number:
55
Not 10, 20 or 30
```

2.3 C++ For Loop

The C++ for loop is used to iterate a part of the program several times. If the number of iteration is fixed, it is recommended to use for loop than while or do-while loops. The C++ for loop is same as C/C. We can initialize variable, check condition and increment/decrement value. The syntax is as follows.

```
for(initialization; condition; incr/decr){
//code to be executed
}
```

C++ For Loop Example

```
#include <iostream>
using namespace std;
int main() {
 for(int i=1;i<=10;i++){
    cout<<i <<"\n";
  }
}
```

Output

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

2.3.1 C++ Nested For Loop

In C++, we can use for loop inside another for loop, it is known as nested for loop. The inner loop is executed fully when outer loop is executed one time. So if outer loop and inner loop are executed 4 times, inner loop will be executed 4 times for each outer loop i.e. total 16 times. Let's see a simple example of nested for loop in C++.

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

```
3
4  int main () {
5  for(int i=1;i<=3;i++){
6    for(int j=1;j<=3;j++){
7      cout<<i<<" "<<j<<"\n";
8    }
9  }
10 }
```

Output

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

2.3.2 C++ Infinite For Loop

If we use double semicolon in for loop, it will be executed infinite times. Let's see a simple example of infinite for loop in C++.

```
1 #include <iostream>
2 using namespace std;
3
4 int main () {
5 for (; ;)
6 {
7   cout<<"Infinitive For Loop";
8 }
9 }
```

2.4 C++ While loop

In C++, while loop is used to iterate a part of the program several times. If the number of iteration is not fixed, it is recommended to use while loop than for loop.

```
1 while(condition){
2 //code to be executed
3 }
```

Let's see a simple example of while loop to print table of 1.

```
#include <iostream>
using namespace std;
int main() {
int i=1;
while(i<=10)
{
  cout<<i <<"\n";
  i++;
}
}
```

Ouput

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

2.4.1 C++ Nested While Loop Example

In C++, we can use while loop inside another while loop, it is known as nested while loop. The nested while loop is executed fully when outer loop is executed once. Let's see a simple example of nested while loop in C++ programming language.

```
#include <iostream>
using namespace std;
int main () {
int i=1;
while(i<=3)
{
  int j = 1;
  while (j <= 3)
  {
    cout<<i<<" "<<j<<"\n";
    j++;
  }
i++;
}
}
```

Output:

```
1 1
1 2
1 3
2 1
2 2
2 3
3 1
3 2
3 3
```

2.4.2 C++ Infinitive While Loop

We can also create infinite while loop by passing true as the test condition.

```
#include <iostream>
using namespace std;
int main () {
while(true)
{
  cout<<"Infinitive While Loop";
}
}
```

2.4.3 C++ Do-While Loop

The C++ do-while loop is used to iterate a part of the program several times. If the number of iteration is not fixed and you must have to execute the loop at least once, it is recommended to use do-while loop. The C++ do-while loop is executed at least once because condition is checked after loop body.

```
do{
//code to be executed
}while(condition);
```

Let's see a simple example of C++ do-while loop to print the table of 1.

```
#include <iostream>
using namespace std;
int main() {
int i = 1;
do{
```

```
  cout<<i<<"\n";
  i++;
} while (i <= 10) ;
}
```

Output:

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

2.4.4 C++ Nested do-while Loop

In C++, if you use do-while loop inside another do-while loop, it is known as nested do-while loop. The nested do-while loop is executed fully for each outer do-while loop. Let's see a simple example of nested do-while loop in C++.

```
#include <iostream>
using namespace std;
int main() {
int i = 1;
do{
  int j = 1;
  do{
    cout<<i<<"\n";
    j++;
  } while (j <= 3) ;
  i++;
} while (i <= 3) ;
}
```

Output:

```
1 1
1 2
1 3
2 1
2 2
2 3
```

```
3 1
3 2
3 3
```

2.4.5 C++ Infinitive do-while Loop

In C++, if you pass true in the do-while loop, it will be infinitive do-while loop.

```
do{
//code to be executed
}while(true);
```

C++ Infinitive do-while Loop Example

```
#include <iostream>
using namespace std;
int main() {
do{
  cout<<"Infinitive do-while Loop";
  } while(true);
}
```

2.5 C++ Break & Continue Statement

The C++ break is used to break loop or switch statement. It breaks the current flow of the program at the given condition. In case of inner loop, it breaks only inner loop.

```
jump-statement;
break;
```

Let's see a simple example of C++ break statement which is used inside the loop.

```
#include <iostream>
using namespace std;
int main() {
for (int i = 1; i <= 10; i++)
  {
    if (i == 5)
    {
      break;
    }
  cout<<i<<"\n";
```

```
  }
}
```

Output:

```
1
2
3
4
```

2.5.1 C++ Break Statement with Inner Loop

The C++ break statement breaks inner loop only if you use break statement inside the inner loop. Let's see the example code:

```
#include <iostream>
using namespace std;
int main()
{
for(int i=1;i<=3;i++){
  for(int j=1;j<=3;j++){
    if(i==2&&j==2){
      break;
    }
      cout<<i<<" "<<j<<"\n";
  }
}
}
```

Output:

```
1 1
1 2
1 3
2 1
3 1
3 2
3 3
```

2.5.2 C++ Continue Statement

The C++ continue statement is used to continue loop. It continues the current flow of the program and skips the remaining code at specified condition. In case of inner loop, it continues only inner loop.

```
jump-statement;
continue;
```

C++ Continue Statement Example

```
#include <iostream>
using namespace std;
int main()
{
 for(int i=1;i<=10;i++){
  if(i==5){
    continue;
  }
  cout<<i<<"\n";
 }
}
```

Output:

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

2.5.3 C++ Continue Statement with Inner Loop

C++ Continue Statement continues inner loop only if you use continue statement inside the inner loop.

```
#include <iostream>
using namespace std;
int main()
{
 for(int i=1;i<=3;i++){
  for(int j=1;j<=3;j++){
   if(i==2&&j==2){
    continue;
   }
  cout<<i<<" "<<j<<"\n";
  }
 }
}
```

Output:

```
1 1
1 2
1 3
2 1
2 3
3 1
3 2
3 3
```

2.5.4 C++ Goto Statement

The C++ goto statement is also known as jump statement. It is used to transfer control to the other part of the program. It unconditionally jumps to the specified label. It can be used to transfer control from deeply nested loop or switch case label. Let's see the simple example of goto statement in C++.

```
#include <iostream>
using namespace std;
int main()
{
ineligible:
  cout<<"You are not eligible to vote!\n";
  cout<<"Enter your age:\n";
  int age;
  cin>>age;
  if (age < 18){
    goto ineligible;
  }
  else
  {
    cout<<"You are eligible to vote!";
  }
}
```

Output:

```
You are not eligible to vote!
Enter your age:
16
You are not eligible to vote!
Enter your age:
7
You are not eligible to vote!
Enter your age:
```

```
22
You are eligible to vote!
```

2.6 Arrays

Like other programming languages, array in C++ is a group of similar types of elements that have contiguous memory location. In C++ `std::array` is a container that encapsulates fixed size arrays. In C++, array index starts from 0. We can store only fixed set of elements in C++ array.

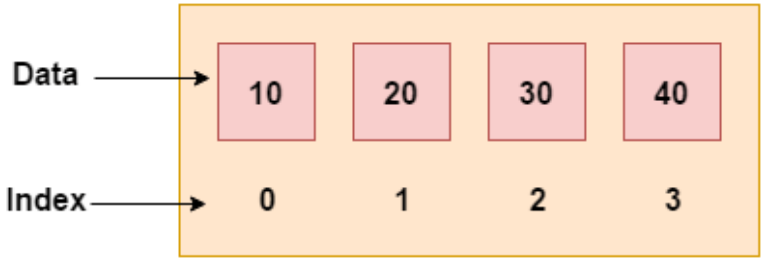


Figure 2.1: Arrays in C++

2.6.1 Advantages of C++ Array

1. Code Optimization (less code)
2. Random Access
3. Easy to traverse data
4. Easy to manipulate data
5. Easy to sort data etc.

However, there one potential disadvantage of an array is that they are *fixed size data structures*.

2.6.2 C++ Array Types

There are 2 types of arrays in C++ programming:

1. One Dimensional Array
2. Multidimensional Array

2.6.3 One Dimensional Array

Let's see a simple example of C++ array, where we are going to create, initialize and traverse array.

```
#include <iostream>
using namespace std;
int main()
{
 int arr[5]={10, 0, 20, 0, 30};  //creating and initializing
     ↪  array
  //traversing array
  for (int i = 0; i < 5; i++)
  {
    cout<<arr[i]<<"\n";
  }
}
```

2.7 Multidimensional arrays

The multidimensional array is also known as rectangular arrays in C++. It can be two dimensional or three dimensional. The data is stored in tabular form (row $*$ column) which is also known as matrix. Let's see a simple example of multidimensional array in C++ which declares, initializes and traverse two dimensional arrays.

```
#include <iostream>
using namespace std;
int main()
{
  int test[3][3];  //declaration of 2D array
  test[0][0]=5;  //initialization
  test[0][1]=10;
  test[1][1]=15;
  test[1][2]=20;
  test[2][0]=30;
  test[2][2]=10;
  //traversal
  for(int i = 0; i < 3; ++i)
  {
```

```
  for(int j = 0; j < 3; ++j)
  {
    cout<< test[i][j]<<" ";
  }
  cout<<"\n"; //new line at each row
  }
  return 0;
}
```

Output:

```
5 10 0
0 15 20
30 0 10
```

Declaration and initialization at same time

Let's see a simple example of multidimensional array which initializes array at the time of declaration.

```
#include <iostream>
using namespace std;
int main()
{
int test[3][3] =
{
  {2, 5, 5},
  {4, 0, 3},
  {9, 1, 8}  };  //declaration and initialization
//traversal
for(int i = 0; i < 3; ++i)
{
  for(int j = 0; j < 3; ++j)
  {
      cout<< test[i][j]<<" ";
  }
  cout<<"\n"; //new line at each row
}
return 0;
}
```

2.8 Functions in C/C++

A function is a set of statements that take inputs, do some specific computation and produces output. The idea is to put some commonly or repeatedly done task together and make a function so

that instead of writing the same code again and again for different inputs, we can call the function.

The general form of a function is:

```
return_type function_name([ arg1_type arg1_name, ... ]) {
    ↪ code }
```

Example:

Below is a simple C/C++ program to demonstrate functions.

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

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

int main() {
  int a = 10, b = 20;

  // Calling above function to find max of 'a' and 'b'
  int m = max(a, b);

  cout << "m is " << m;
  return 0;
}
```

Output:

```
m is 20
```

2.8.1 Why do we need functions?

1. Functions help us in reducing code redundancy. If functionality is performed at multiple places in software, then rather than writing the same code, again and again, we create a function and call it everywhere. This also helps in maintenance as we have to change at one place if we make future changes to the functionality.

2. Functions make code modular. Consider a big file having many lines of code. It becomes really simple to read and use the code if the code is divided into functions.

3. Functions provide abstraction. For example, we can use library functions without worrying about their internal working.

2.8.2 Function Declaration

A function declaration tells the compiler about the number of parameters function takes, data-types of parameters, and return type of function. Putting parameter names in function declaration is optional in the function declaration, but it is necessary to put them in the definition. Below are an example of function declarations.

```
// A function that takes two integers as parameters
// and returns an integer
int max(int, int);

// A function that takes an int pointer and an int variable
    ↪ as parameters
// and returns a pointer of type int
int *swap(int*,int);

// A function that takes a charas parameters
// and returns an reference variable
char *call(char b);

// A function that takes a char and an int as parameters
// and returns an integer
int fun(char, int);
```

2.8.3 Return From Void Functions

Void functions are known as Non-Value Returning functions. They are "void" due to the fact that they are not supposed to return values. We cannot return values but there is something we can surely return from void functions. Void functions do not have a return type, but they can do return values. Some of the cases are listed below:

1. *A Void Function Can Return*: We can simply write a return statement in a void fun(). In fact, it is considered a good

practice (for readability of code) to write a return; statement to indicate the end of the function.

```
// CPP Program to demonstrate void functions
#include <iostream>
using namespace std;

void fun()
{
  cout << "Hello";

  // We can write return in void
  return;
}

// Driver Code
int main()
{
  fun();
  return 0;
}
```

2. *A void fun() can return another void function*: A void function can also call another void function while it is terminating. For example,

```
// C++ code to demonstrate void()
// returning void()
#include <iostream>
using namespace std;

// A sample void function
void work()
{
  cout << "The void function has returned "
          " a void() !!! \n";
}

// Driver void() returning void work()
void test()
{
  // Returning void function
  return work();
}

// Driver Code
int main()
{
  // Calling void function
```

```
  test();
  return 0;
}
```

3. *A void() can return a void value*: A *void()* cannot return a value that can be used. But it can return a value that is void without giving an error. For example,

```
// C++ code to demonstrate void()
// returning a void value
#include <iostream>
using namespace std;

// Driver void() returning a void value
void test()
{
  cout << "Hello";

  // Returning a void value
  return (void)"Doesn't Print";
}

// Driver Code
int main()
{
  test();
  return 0;
}
```

2.8.4 Parameter Passing to functions

The parameters passed to function are called *actual parameters*. For example, in the above program 10 and 20 are actual parameters. The parameters received by function are called *formal parameters*. For example, in the above program x and y are formal parameters. There are two most popular ways to pass parameters.

1. Pass by Value: In this parameter passing method, values of actual parameters are copied to function's formal parameters and the two types of parameters are stored in different memory locations. So any changes made inside functions are not reflected in actual parameters of caller.

2. Pass by Reference Both actual and formal parameters refer to same locations, so any changes made inside the function

are actually reflected in actual parameters of caller.

Parameters are always passed by value in C. For example. in the below code, value of x is not modified using the function *fun()*.

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

void fun(int x) {
  x = 30;
}

int main() {
  int x = 20;
  fun(x);
  cout << "x = " << x;
  return 0;
}
```

Output:

```
x = 20
```

However, in C, we can use pointers to get the effect of pass-by reference. For example, consider the below program. The function *fun()* expects a pointer ptr to an integer (or an address of an integer). It modifies the value at the address ptr. The dereference operator * is used to access the value at an address. In the statement "*ptr = 30", value at address *ptr* is changed to 30. The address operator & is used to get the address of a variable of any data type. In the function call statement "fun(&x)", the address of x is passed so that x can be modified using its address.

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

void fun(int *ptr)
{
  *ptr = 30;
}

int main() {
  int x = 20;
  fun(&x);
  cout << "x = " << x;

  return 0;
}
```

Output:

```
x = 30
```

2.9 Pointers

Pointers are symbolic representation of addresses. They enable programs to simulate call-by-reference as well as to create and manipulate dynamic data structures. The pointer in C++ language is a variable, it is also known as locator or indicator that points to an address of a value.

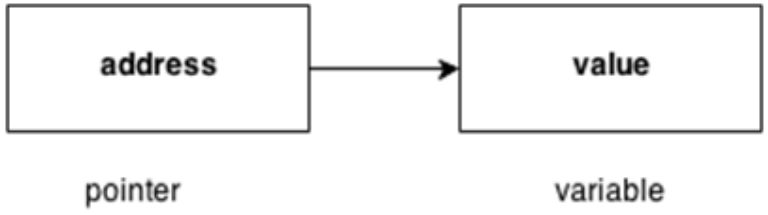


Figure 2.2: Pointers

2.9.1 Advantage of pointer

1. Pointer reduces the code and improves the performance, it is used to retrieving strings, trees etc. and used with arrays, structures and functions.
2. We can return multiple values from function using pointer.
3. It makes you able to access any memory location in the computer's memory.
4. *Dynamic memory allocation*: We can dynamically allocate memory using `malloc()` and `calloc()` functions where pointer is used.
5. *Arrays, Functions and Structures*: Pointers in c language are widely used in arrays, functions and structures. It reduces the code and improves the performance.

2.9.2 Usage of pointer

There are many usage of pointers in C++ language.

1. Define a pointer variable
2. Assigning the address of a variable to a pointer using unary operator () which returns the address of that variable.
3. Accessing the value stored in the address using unary operator (*) which returns the value of the variable located at the address specified by its operand.

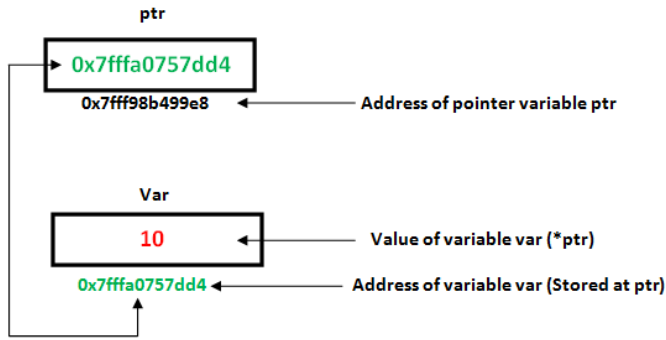


Figure 2.3: Declaring pointers

2.9.3 Symbols used in pointer

Symbol	Name	Description
& (ampersand sign)	Address operator	Determine the address of a variable.
* (asterisk sign)	Indirection operator	Access the value of an address.

Table 2.1: Symbols used for pointers

2.9.4 Declaring a pointer

The pointer in C++ language can be declared using ∗ (asterisk symbol).

```
int *   a; //pointer to int
char *  c; //pointer to char
```

2.9.5 Pointer Example

Let's see the simple example of using pointers printing the address and value.

```
#include <iostream>
using namespace std;
int main()  {
  int number=30;
  int *   p;
  p=&number;//stores the address of number variable
  cout<<"Address of number variable is:"<<&number<<endl;
  cout<<"Address of p variable is:"<<p<<endl;
  cout<<"Value of p variable is:"<<*p<<endl;
  return 0;
}
```

Output:

```
Address of number variable is:0x7ffccc8724c4
Address of p variable is:0x7ffccc8724c4
Value of p variable is:30
```

The other example using functions

```
// C++ program to illustrate Pointers in C++

#include <bits/stdc++.h>
using namespace std;
void pointer_test()
{
  int var = 20;

      //declare pointer variable
  int *ptr;

  //note that data type of ptr and var must be same
  ptr = &var;
```

```
14
15    // assign the address of a variable to a pointer
16    cout << "Value at ptr = " << ptr << "\n";
17    cout << "Value at var = " << var << "\n";
18    cout << "Value at *ptr = " << *ptr << "\n";
19  }
20  //Driver program
21  int main()
22  {
23    pointer_test();
24  }
```

Output

```
Value at ptr = 0x7ffcb9e9ea4c
Value at var = 20
Value at *ptr = 20
```

2.9.6 Pointer Expressions and Pointer Arithmetic

A limited set of arithmetic operations can be performed on pointers which are:

1. incremented (++)
2. decremented (—)
3. an integer may be added to a pointer (+ or +=)
4. an integer may be subtracted from a pointer (– or -=)
5. difference between two pointers (p1-p2)

Pointer arithmetic on arrays using loops

```
1  // C++ program to illustrate Pointer Arithmetic in C++
2  #include <bits/stdc++.h>
3  using namespace std;
4  void pointer_in_loops() {
5    //Declare an array
6    int v[3] = {10, 100, 200};
7
8    //declare pointer variable
9    int *ptr;
10
11   //Assign the address of v[0] to ptr
12   ptr = v;
13
```

```
  for (int i = 0; i < 3; i++) {
    cout << "Value at ptr = " << ptr << "\n";
    cout << "Value at *ptr = " << *ptr << "\n";

    // Increment pointer ptr by 1
    ptr++;
  }
}

//Driver program
int main()
{
  pointer_in_loops();
}
```

Output

```
Output:
Value at ptr = 0x7fff9a9e7920
Value at *ptr = 10
Value at ptr = 0x7fff9a9e7924
Value at *ptr = 100
Value at ptr = 0x7fff9a9e7928
Value at *ptr = 200
```

2.9.7 References and Pointers

There are 3 ways to pass C++ arguments to a function:

1. call-by-value
2. call-by-reference with pointer argument
3. call-by-reference with reference argument

```
// C++ program to illustrate call-by-methods in C++

#include <bits/stdc++.h>
using namespace std;
//Pass-by-Value
int square1(int n)
{
  //Address of n in square1() is not the same as n1 in main
      ↪ ()
  cout << "address of n1 in square1(): " << &n << "\n";

  // clone modified inside the function
```

```
12   n *= n;
13   return n;
14 }
15 //Pass-by-Reference with Pointer Arguments
16 void square2(int *n)
17 {
18   //Address of n in square2() is the same as n2 in main()
19   cout << "address of n2 in square2(): " << n << "\n";
20
21   // Explicit de-referencing to get the value pointed-to
22   *n *= *n;
23 }
24 //Pass-by-Reference with Reference Arguments
25 void square3(int &n)
26 {
27   //Address of n in square3() is the same as n3 in main()
28   cout << "address of n3 in square3(): " << &n << "\n";
29
30   // Implicit de-referencing (without '*')
31   n *= n;
32 }
33 void Main()
34 {
35   //Call-by-Value
36   int n1=8;
37   cout << "address of n1 in main(): " << &n1 << "\n";
38   cout << "Square of n1: " << square1(n1) << "\n";
39   cout << "No change in n1: " << n1 << "\n";
40
41   //Call-by-Reference with Pointer Arguments
42   int n2=8;
43   cout << "address of n2 in main(): " << &n2 << "\n";
44   square2(&n2);
45   cout << "Square of n2: " << n2 << "\n";
46   cout << "Change reflected in n2: " << n2 << "\n";
47
48   //Call-by-Reference with Reference Arguments
49   int n3=8;
50   cout << "address of n3 in main(): " << &n3 << "\n";
51   square3(n3);
52   cout << "Square of n3: " << n3 << "\n";
53   cout << "Change reflected in n3: " << n3 << "\n";
54
55
56 }
57 //Driver program
58 int main()
59 {
60   Main();
```

```
}
```

Output

```
address of n1 in main(): 0x7ffcdb2b4a44
address of n1 in square1(): 0x7ffcdb2b4a2c
Square of n1: 64
No change in n1: 8
address of n2 in main(): 0x7ffcdb2b4a48
address of n2 in square2(): 0x7ffcdb2b4a48
Square of n2: 64
Change reflected in n2: 64
address of n3 in main(): 0x7ffcdb2b4a4c
address of n3 in square3(): 0x7ffcdb2b4a4c
Square of n3: 64
Change reflected in n3: 64
```

Chapter 3

C++ OOPs Concepts

The major purpose of C++ programming is to introduce the concept of object orientation to the C programming language. Object Oriented Programming is a paradigm that provides many concepts such as inheritance, data binding, polymorphism etc. The programming paradigm where everything is represented as an object is known as truly object-oriented programming language.

3.0.1 OOPs (Object Oriented Programming System)

Object means a real word entity such as pen, chair, table etc. Object-Oriented Programming is a methodology or paradigm to design a program using classes and objects. It simplifies the software development and maintenance by providing some concepts:

1. Object
2. Class
3. Inheritance
4. Polymorphism
5. Abstraction
6. Encapsulation

Object

Any entity that has state and behavior is known as an object. For example: chair, pen, table, keyboard, bike etc. It can be physical and logical.

Class

Collection of objects is called class. It is a logical entity.

Inheritance

When one object acquires all the properties and behaviours of parent object i.e. known as inheritance. It provides code reusability. It is used to achieve runtime polymorphism.

Polymorphism

When one task is performed by different ways i.e. known as polymorphism. For example: to convince the customer differently, to draw something e.g. shape or rectangle etc. In C++, we use Function overloading and Function overriding to achieve polymorphism.

Abstraction

Hiding internal details and showing functionality is known as abstraction. For example: phone call, we don't know the internal processing. In C++, we use abstract class and interface to achieve abstraction.

Encapsulation

Binding (or wrapping) code and data together into a single unit is known as encapsulation. For example: capsule, it is wrapped with different medicines.

3.0.2 Advantage of OOPs over Procedure-oriented programming language

1. OOPs makes development and maintenance easier where as in Procedure-oriented programming language it is not easy

to manage if code grows as project size grows.

2. OOPs provide data hiding whereas in Procedure-oriented programming language a global data can be accessed from anywhere.

3. OOPs provide ability to simulate real-world event much more effectively. We can provide the solution of real word problem if we are using the Object-Oriented Programming language.

3.1 C++ Object and Class

Since C++ is an object-oriented language, program is designed using objects and classes in C++. In C++, Object is a real world entity, for example, chair, car, pen, mobile, laptop etc. In other words, object is an entity that has state and behavior. Here, state means data and behavior means functionality. Object is a runtime entity, it is created at runtime. Object is an instance of a class. All the members of the class can be accessed through object. Let's see an example to create object of student class using s1 as the reference variable.

```
Student s1;  //creating an object of Student
```

In this example, Student is the type and s1 is the reference variable that refers to the instance of Student class.

3.1.1 C++ Class

In C++, class is a group of similar objects. It is a template from which objects are created. It can have fields, methods, constructors etc. Let's see an example of C++ class that has three fields only.

```
1 class Student
2  {
3      public:
4      int id;  //field or data member
5      float salary; //field or data member
6      String name;//field or data member
7  }
```

C++ Object and Class Example

Let's see an example of class that has two fields: *id* and *name*. It creates instance of the class, initializes the object and prints the object value.

```
#include <iostream>
using namespace std;
class Student {
   public:
      int id;//data member (also instance variable)
      string name;//data member(also instance variable)
};
int main() {
    Student s1; //creating an object of Student
    s1.id = 201;
    s1.name = "Sonoo Jaiswal";
    cout<<s1.id<<endl;
    cout<<s1.name<<endl;
    return 0;
}
```

Output:

```
201
Sonoo Jaiswal
```

3.1.2 C++ Class Example: Initialize and Display data through method

Let's see another example of C++ class where we are initializing and displaying object through method.

```
#include <iostream>
using namespace std;
class Student {
   public:
       int id;//data member (also instance variable)
       string name;//data member(also instance variable)
       void insert(int i, string n)
        {
            id = i;
            name = n;
        }
       void display()
        {
            cout<<id<<"  "<<name<<endl;
```

```
        }
};
int main(void) {
    Student s1; //creating an object of Student
    Student s2; //creating an object of Student
    s1.insert(201, "Sonoo");
    s2.insert(202, "Nakul");
    s1.display();
    s2.display();
    return 0;
}
```

Output:

```
201  Sonoo
202  Nakul
```

3.1.3 C++ Class Example: Store and Display Employee Information

Let's see another example of C++ class where we are storing and displaying employee information using method.

```
#include <iostream>
using namespace std;
class Employee {
   public:
       int id;//data member (also instance variable)
       string name;//data member(also instance variable)
       float salary;
       void insert(int i, string n, float s)
        {
            id = i;
            name = n;
            salary = s;
        }
       void display()
        {
            cout<<id<<"  "<<name<<"  "<<salary<<endl;
        }
};
int main(void) {
    Employee e1; //creating an object of Employee
    Employee e2; //creating an object of Employee
    e1.insert(201, "Sonoo",990000);
    e2.insert(202, "Nakul", 29000);
    e1.display();
```

```
    e2.display();
    return 0;
}
```

Output:

```
201  Sonoo  990000
202  Nakul  29000
```

3.2 C++ Constructor

In C++, constructor is a special method which is invoked automatically at the time of object creation. It is used to initialize the data members of new object generally. The constructor in C++ has the same name as class or structure. There can be two types of constructors in C++.

1. Default constructor
2. Parameterized constructor

3.2.1 C++ Default Constructor

A constructor which has no argument is known as default constructor. It is invoked at the time of creating object. Let's see the simple example of C++ default Constructor.

```
#include <iostream>
using namespace std;
class Employee
 {
   public:
    Employee()
    {
      cout<<"Default Constructor Invoked"<<endl;
    }
};
int main(void)
{
  Employee e1; //creating an object of Employee
  Employee e2;
  return 0;
}
```

Output:

```
Default Constructor Invoked
Default Constructor Invoked
```

3.2.2 C++ Parameterized Constructor

A constructor which has parameters is called parameterized constructor. It is used to provide different values to distinct objects. Let's see the simple example of C++ Parameterized Constructor.

```
#include <iostream>
using namespace std;
class Employee {
 public:
   int id;//data member (also instance variable)
   string name;//data member(also instance variable)
   float salary;
   Employee(int i, string n, float s)
    {
        id = i;
        name = n;
        salary = s;
    }
   void display()
    {
        cout<<id<<"  "<<name<<"  "<<salary<<endl;
    }
};
int main(void) {
  Employee e1 =Employee(101, "Sonoo", 890000); //creating an
      ↪  object of Employee
  Employee e2=Employee(102, "Nakul", 59000);
  e1.display();
  e2.display();
  return 0;
}
```

Output:

```
101  Sonoo  890000
102  Nakul  59000
```

3.3 C++ Destructor

A destructor works opposite to constructor; it destructs the objects of classes. It can be defined only once in a class. Like constructors,

it is invoked automatically. A destructor is defined like constructor. It must have same name as class. But it is prefixed with a tilde sign ( ).

C++ Constructor and Destructor Example

Let's see an example of constructor and destructor in C++ which is called automatically.

```
#include <iostream>
using namespace std;
class Employee
 {
   public:
    Employee()
    {
      cout<<"Constructor Invoked"<<endl;
    }
    ~Employee()
    {
      cout<<"Destructor Invoked"<<endl;
    }
};
int main(void)
{
  Employee e1; //creating an object of Employee
  Employee e2; //creating an object of Employee
  return 0;
}
```

Output:

```
Constructor Invoked
Constructor Invoked
Destructor Invoked
Destructor Invoked
```

Destructor example: string operations

```
class String {
private:
  char* s;
  int size;

public:
  String(char*); // constructor
```

```
  ~String(); // destructor
};

String::String(char* c)
{
  size = strlen(c);
  s = new char[size + 1];
  strcpy(s, c);
}
String::~String() { delete[] s; }
```

3.4 C++ this Pointer

In C++ programming, **this** is a keyword that refers to the current instance of the class. There can be 3 main usage of *this* keyword in C++.

1. It can be used to pass current object as a parameter to another method.
2. It can be used to refer current class instance variable.
3. It can be used to declare indexers.

C++ this Pointer Example

Let's see the example of this keyword in C++ that refers to the fields of current class.

```
#include <iostream>
using namespace std;
class Employee {
 public:
   int id; //data member (also instance variable)
   string name; //data member(also instance variable)
   float salary;
   Employee(int id, string name, float salary)
    {
      this->id = id;
      this->name = name;
      this->salary = salary;
    }
   void display()
    {
      cout<<id<<"  "<<name<<"  "<<salary<<endl;
    }
```

```
};
int main(void) {
    Employee e1 =Employee(101, "Sonoo", 890000); //creating
        ↪ an object of Employee
    Employee e2=Employee(102, "Nakul", 59000); //creating an
        ↪  object of Employee
    e1.display();
    e2.display();
    return 0;
}
```

Output:

```
101  Sonoo  890000
102  Nakul  59000
```

3.5 C++ static

In C++, `static` is a keyword or modifier that belongs to the type not instance. So instance is not required to access the static members. In C++, static can be field, method, constructor, class, properties, operator and event. ***static* keyword is used to achieve memory efficiency in programming**.

3.5.1 C++ Static Field

A field which is declared as static is called static field. Unlike instance field which gets memory each time whenever you create object, there is only one copy of static field created in the memory. It is shared to all the objects. It is used to refer the common property of all objects such as rateOfInterest in case of Account, companyName in case of Employee etc.

Let's see the simple example of static field in C++.

```
#include <iostream>
using namespace std;
class Account {
 public:
   int accno; //data member (also instance variable)
   string name; //data member(also instance variable)
   static float rateOfInterest;
   Account(int accno, string name)
    {
```

```
      this->accno = accno;
      this->name = name;
    }
   void display()
    {
      cout<<accno<< "<<name<< " "<<rateOfInterest<<endl;
    }
};
float Account::rateOfInterest=6.5;
int main(void) {
    Account a1 =Account(201, "Sanjay"); //creating an object
        ↪ of Employee
    Account a2=Account(202, "Nakul"); //creating an object
        ↪ of Employee
    a1.display();
    a2.display();
    return 0;
}
```

Output:

```
201 Sanjay 6.5
202 Nakul 6.5
```

Let's see another example of static keyword in C++ which counts the objects.

```
#include <iostream>
using namespace std;
class Account {
 public:
   int accno; //data member (also instance variable)
   string name;
   static int count;
   Account(int accno, string name)
    {
      this->accno = accno;
      this->name = name;
      count++;
    }
   void display()
    {
      cout<<accno<<" "<<name<<endl;
    }
};
int Account::count=0;
int main(void) {
  Account a1 =Account(201, "Sanjay"); //creating an object
      ↪ of Account
```

```
22   Account a2=Account(202, "Nakul");
23    Account a3=Account(203, "Ranjana");
24   a1.display();
25   a2.display();
26   a3.display();
27   cout<<"Total Objects are: "<<Account::count;
28   return 0;
29 }
```

Output:

```
201 Sanjay
202 Nakul
203 Ranjana
Total Objects are: 3
```

3.6 C++ Inheritance

In C++, inheritance is a process in which one object acquires all the properties and behaviors of its parent object automatically. In such way, you can reuse, extend or modify the attributes and behaviors which are defined in other class.

In C++, the class which inherits the members of another class is called derived class and the class whose members are inherited is called base class. The derived class is the specialized class for the base class. Main purpose of inheritance is *Code reusability*.

3.6.1 Types Of Inheritance

C++ supports five types of inheritance:

1. Single inheritance
2. Multiple inheritance
3. Hierarchical inheritance
4. Multilevel inheritance
5. Hybrid inheritance

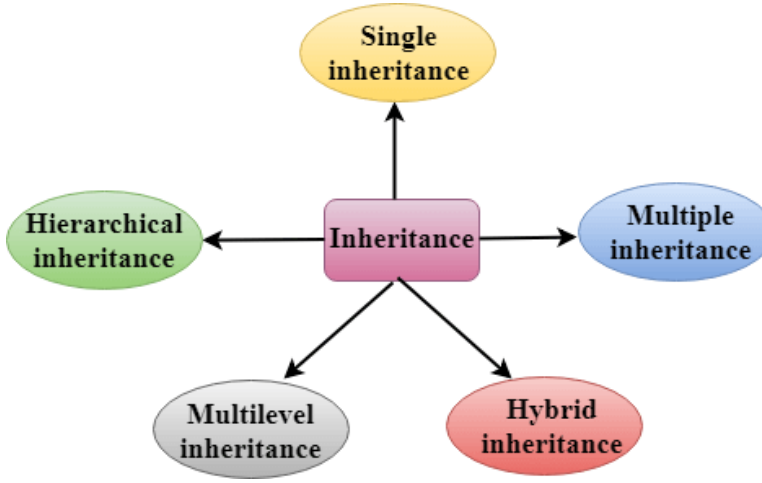


Figure 3.1: Types of inheritance in C++

Derived Classes

A Derived class is defined as the class derived from the base class.

The Syntax of Derived class:

```
class derived_class_name :: visibility-mode base_class_name
{
    // body of the derived class.
}
```

Where,

derived_class_name: It is the name of the derived class.

visibility mode: The visibility mode specifies whether the features of the base class are publicly inherited or privately inherited. It can be public or private.

base_class_name: It is the name of the base class.

When the base class is privately inherited by the derived class, public members of the base class becomes the private members of the derived class. Therefore, the public members of the base class

are not accessible by the objects of the derived class only by the member functions of the derived class.

When the base class is publicly inherited by the derived class, public members of the base class also become the public members of the derived class. Therefore, the public members of the base class are accessible by the objects of the derived class as well as by the member functions of the base class. In C++, the default mode of visibility is private. ***The private members of the base class are never inherited.***

3.6.2 C++ Single Inheritance

Single inheritance is defined as the inheritance in which a derived class is inherited from the only one base class.

Where 'A' is the base class, and 'B' is the derived class.

C++ Single Level Inheritance Example: Inheriting Fields

When one class inherits another class, it is known as single level inheritance. Let's see the example of single level inheritance which inherits the fields only.

```
#include <iostream>
using namespace std;
 class Account {
   public:
   float salary = 60000;
 };
   class Programmer: public Account {
   public:
   float bonus = 5000;
   };
int main(void) {
   Programmer p1;
   cout<<"Salary: "<<p1.salary<<endl;
   cout<<"Bonus: "<<p1.bonus<<endl;
  return 0;
}
```

Output:

```
Salary: 60000
Bonus: 5000
```

In the above example, Employee is the base class and Programmer is the derived class.

C++ Single Level Inheritance Example: Inheriting Methods

Let's see another example of inheritance in C++ which inherits methods only.

```
#include <iostream>
using namespace std;
 class Animal {
   public:
 void eat() {
    cout<<"Eating..."<<endl;
 }
   };
   class Dog: public Animal
   {
       public:
     void bark(){
    cout<<"Barking...";
     }
   };
int main(void) {
    Dog d1;
    d1.eat();
    d1.bark();
    return 0;
}
```

Output:

```
Eating...
Barking...
```

Let's see a simple example.

```
#include <iostream>
using namespace std;
class A
{
    int a = 4;
    int b = 5;
    public:
    int mul()
    {
```

```
10          int c = a*b;
11          return c;
12      }
13  };
14
15  class B : private A
16  {
17      public:
18      void display()
19      {
20          int result = mul();
21          std::cout <<"Multiplication of a and b is : "<<
                ↪ result<< std::endl;
22      }
23  };
24  int main()
25  {
26     B b;
27     b.display();
28
29      return 0;
30  }
```

Output:

```
Multiplication of a and b is : 20
```

In the above example, class A is privately inherited. Therefore, the mul() function of class 'A' cannot be accessed by the object of class B. It can only be accessed by the member function of class B.

How to make a Private Member Inheritable

The private member is not inheritable. If we modify the visibility mode by making it public, but this takes away the advantage of data hiding.

C++ introduces a third visibility modifier, i.e., protected. The member which is declared as protected will be accessible to all the member functions within the class as well as the class immediately derived from it. Visibility modes can be classified into three categories:

1. *Public*: When the member is declared as public, it is accessible to all the functions of the program.

Base class visibility	Derived class visibility		
	Public	Private	Protected
Private	Not Inherited	Not Inherited	Not Inherited
Protected	Protected	Private	Protected
Public	Public	Private	Protected

Table 3.1: Visibility of Inherited Members

2. *Private*: When the member is declared as private, it is accessible within the class only.
3. *Protected*: When the member is declared as protected, it is accessible within its own class as well as the class immediately derived from it.

Visibility of Inherited Members

3.6.3 C++ Multilevel Inheritance

Multilevel inheritance is a process of deriving a class from another derived class.

C++ Multi Level Inheritance Example

When one class inherits another class which is further inherited by another class, it is known as multi level inheritance in C++. Inheritance is transitive so the last derived class acquires all the members of all its base classes. Let's see the example of multi level inheritance in C++.

```
#include <iostream>
using namespace std;
 class Animal {
   public:
 void eat() {
    cout<<"Eating..."<<endl;
 }
   };
   class Dog: public Animal
   {
       public:
     void bark(){
    cout<<"Barking..."<<endl;
```

```
    }
  };
  class BabyDog: public Dog
  {
      public:
    void weep() {
   cout<<"Weeping...";
    }
  };
int main(void) {
    BabyDog d1;
    d1.eat();
    d1.bark();
     d1.weep();
     return 0;
}
```

Output:

```
Eating...
Barking...
Weeping...
```

Multiple inheritance is the process of deriving a new class that inherits the attributes from two or more classes.

Syntax of the Derived class:

```
class D : visibility B-1, visibility B-2, ?
{
    // Body of the class;
}
```

Let's see a simple example of multiple inheritance.

```
#include <iostream>
using namespace std;
class A
{
    protected:
     int a;
    public:
    void get_a(int n)
    {
        a = n;
    }
};

```

```
class B
{
    protected:
    int b;
    public:
    void get_b(int n)
    {
        b = n;
    }
};
class C : public A,public B
{
   public:
    void display()
    {
        std::cout << "The value of a is : " <<a<< std::endl;
        std::cout << "The value of b is : " <<b<< std::endl;
        cout<<"Addition of a and b is : "<<a+b;
    }
};
int main()
{
   C c;
   c.get_a(10);
   c.get_b(20);
   c.display();

    return 0;
}
```

Output:

```
The value of a is : 10
The value of b is : 20
Addition of a and b is : 30
```

In the above example, class 'C' inherits two base classes 'A' and 'B' in a public mode.

3.6.4 Ambiguity Resolution in Inheritance

Ambiguity can be occurred in using the multiple inheritance when a function with the same name occurs in more than one base class.

Let's understand this through an example:

```
#include <iostream>
using namespace std;
class A
{
    public:
    void display()
    {
        std::cout << "Class A" << std::endl;
    }
};
class B
{
    public:
    void display()
    {
        std::cout << "Class B" << std::endl;
    }
};
class C : public A, public B
{
    void view()
    {
        display();
    }
};
int main()
{
    C c;
    c.display();
    return 0;
}
```

Output:

```
error: reference to 'display' is ambiguous
        display();
```

The above issue can be resolved by using the class resolution operator with the function. In the above example, the derived class code can be rewritten as:

```
class C : public A, public B
{
    void view()
    {
        A :: display();         // Calling the display()
            ↪ function of class A.
```

```
        B :: display();          // Calling the display()
            ↪ function of class B.

    }
};
```

An ambiguity can also occur in single inheritance.

Consider the following situation:

```
class A
{
   public:
void display()
{
   cout<<?Class A?;
}
} ;
class B
{
  public:
 void display()
{
 cout<<?Class B?;
}
} ;
```

In the above case, the function of the derived class overrides the method of the base class. Therefore, call to the display() function will simply call the function defined in the derived class. If we want to invoke the base class function, we can use the class resolution operator.

```
int main()
{
    B b;
   b.display();                // Calling the display()
       ↪ function of B class.
   b.B :: display();       // Calling the display() function
       ↪  defined in B class.
}
```

3.6.5 C++ Hybrid Inheritance

Hybrid inheritance is a combination of more than one type of inheritance.

Let's see a simple example:

```
#include <iostream>
using namespace std;
class A
{
    protected:
    int a;
    public:
    void get_a()
    {
       std::cout << "Enter the value of 'a' : " << std::endl
           ↪ ;
       cin>>a;
    }
};

class B : public A
{
    protected:
    int b;
    public:
    void get_b()
    {
        std::cout << "Enter the value of 'b' : " << std::
            ↪ endl;
       cin>>b;
    }
};
class C
{
    protected:
    int c;
    public:
    void get_c()
    {
        std::cout << "Enter the value of c is : " << std::
            ↪ endl;
        cin>>c;
    }
};

class D : public B, public C
{
    protected:
    int d;
    public:
    void mul()
    {
```

```
45          get_a();
46          get_b();
47          get_c();
48          std::cout << "Multiplication of a,b,c is : " <<a*b*
                ↪ c<< std::endl;
49      }
50  };
51  int main()
52  {
53      D d;
54      d.mul();
55      return 0;
56  }
```

Output:

```
Enter the value of 'a' :
10
Enter the value of 'b' :
20
Enter the value of c is :
30
Multiplication of a,b,c is : 6000
```

3.6.6 C++ Hierarchical Inheritance

Hierarchical inheritance is defined as the process of deriving more than one class from a base class.

Syntax of Hierarchical inheritance:

```
1   class A
2   {
3       // body of the class A.
4   }
5   class B : public A
6   {
7       // body of class B.
8   }
9   class C : public A
10  {
11      // body of class C.
12  }
13  class D : public A
14  {
```

```
    // body of class D.
}
```

Let's see a simple example:

```
#include <iostream>
using namespace std;
class Shape                  // Declaration of base class.
{
    public:
    int a;
    int b;
    void get_data(int n,int m)
    {
        a= n;
        b = m;
    }
};
class Rectangle : public Shape  // inheriting Shape class
{
    public:
    int rect_area()
    {
        int result = a*b;
        return result;
    }
};
class Triangle : public Shape    // inheriting Shape class
{
    public:
    int triangle_area()
    {
        float result = 0.5*a*b;
        return result;
    }
};
int main()
{
    Rectangle r;
    Triangle t;
    int length,breadth,base,height;
    std::cout << "Enter the length and breadth of a
        ↪ rectangle: " << std::endl;
    cin>>length>>breadth;
    r.get_data(length,breadth);
    int m = r.rect_area();
    std::cout << "Area of the rectangle is : " <<m<< std::
        ↪ endl;
    std::cout << "Enter the base and height of the triangle:
```

```
        ↪  " << std::endl;
43      cin>>base>>height;
44      t.get_data(base,height);
45      float n = t.triangle_area();
46      std::cout <<"Area of the triangle is : "  << n<<std::
            ↪ endl;
47      return 0;
48  }
```

Output:

```
Enter the length and breadth of a rectangle:
23
20
Area of the rectangle is : 460
Enter the base and height of the triangle:
2
5
Area of the triangle is : 5
```

3.7 C++ Polymorphism

The term "Polymorphism" is the combination of "poly" + "morphs" which means many forms. It is a greek word. In object-oriented programming, we use 3 main concepts: inheritance, encapsulation, and polymorphism.

Let's consider a real-life example of polymorphism. A lady behaves like a teacher in a classroom, mother or daughter in a home and customer in a market. Here, a single person is behaving differently according to the situations.

There are two types of polymorphism in C++:

1. *Compile time polymorphism*: The overloaded functions are invoked by matching the type and number of arguments. This information is available at the compile time and, therefore, compiler selects the appropriate function at the compile time. It is achieved by function overloading and operator overloading which is also known as static binding or early

binding. Now, let's consider the case where function name and prototype is same.

```
   class A                                    //  base
       ↪ class declaration.
  {
       int a;
       public:
       void display()
       {
             cout<< "Class A ";
        }
  };
class B : public A                          //  derived
    ↪ class declaration.
{
    int b;
    public:
   void display()
  {
        cout<<"Class B";
  }
};
```

In the above case, the prototype of display() function is the same in both the base and derived class. Therefore, the static binding cannot be applied. It would be great if the appropriate function is selected at the run time. This is known as run time polymorphism.

2. *Run time polymorphism*: Run time polymorphism is achieved when the object's method is invoked at the run time instead of compile time. It is achieved by method overriding which is also known as dynamic binding or late binding.

3.7.1 Differences b/w compile time and run time polymorphism.

C++ Runtime Polymorphism Example

Let's see a simple example of run time polymorphism in C++.

```
// an example without the virtual keyword.

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

Compile time polymorphism
The function to be invoked is known at the compile time.
It is also known as overloading, early binding and static binding.
Overloading is a compile time polymorphism where more than one method is having the same name but with the diffe
It is achieved by function overloading and operator overloading.
It provides fast execution as it is known at the compile time.
It is less flexible as mainly all the things execute at the compile time.

Table 3.2: Differences between compile time and runtime polymorphisms

```
class Animal {
    public:
void eat(){
cout<<"Eating...";
    }
};
class Dog: public Animal
{
 public:
 void eat()
    {            cout<<"Eating bread...";
    }
};
int main(void) {
   Dog d = Dog();
   d.eat();
   return 0;
}
```

Output:

```
Eating bread...
```

C++ Run time Polymorphism Example: By using two derived class

Let's see another example of run time polymorphism in C++ where we are having two derived classes.

```
// an example with virtual keyword.

#include <iostream>
using namespace std;
class Shape {                                          //
    ↪ base class
    public:
virtual void draw(){                                // virtual
    ↪ function
```

```
cout<<"drawing..."<<endl;
    }
};
class Rectangle: public Shape                    //
    ↪ inheriting Shape class.
{
 public:
 void draw()
   {
       cout<<"drawing rectangle..."<<endl;
    }
};
class Circle: public Shape                         //
    ↪ inheriting Shape class.

{
 public:
 void draw()
   {
      cout<<"drawing circle..."<<endl;
   }
};
int main(void) {
    Shape *s;                                  //  base class
        ↪ pointer.
    Shape sh;                                  // base class
        ↪ object.
       Rectangle rec;
        Circle cir;
      s=&sh;
     s->draw();
        s=&rec;
     s->draw();
    s=?
    s->draw();
}
```

Output:

```
drawing...
drawing rectangle...
drawing circle...
```

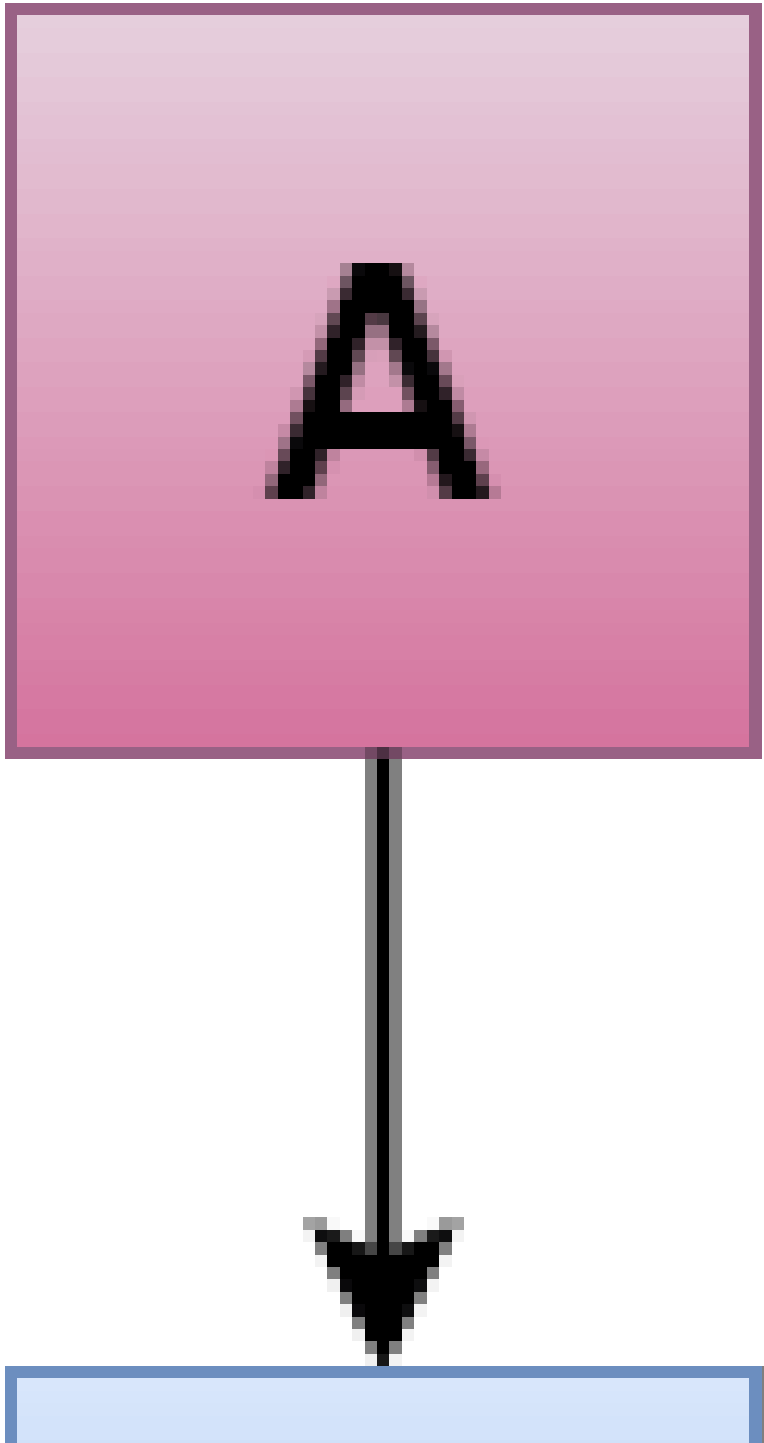
3.7.2 Runtime Polymorphism with Data Members

Runtime Polymorphism can be achieved by data members in C++. Let's see an example where we are accessing the field by reference variable which refers to the instance of derived class.

```
#include <iostream>
using namespace std;
class Animal {                                          //
    ↪ base class declaration.
    public:
    string color = "Black";
};
class Dog: public Animal                              // inheriting
    ↪  Animal class.
{
 public:
    string color = "Grey";
};
int main(void) {
     Animal d= Dog();
    cout<<d.color;
}
```

Output:

```
Black
```


A

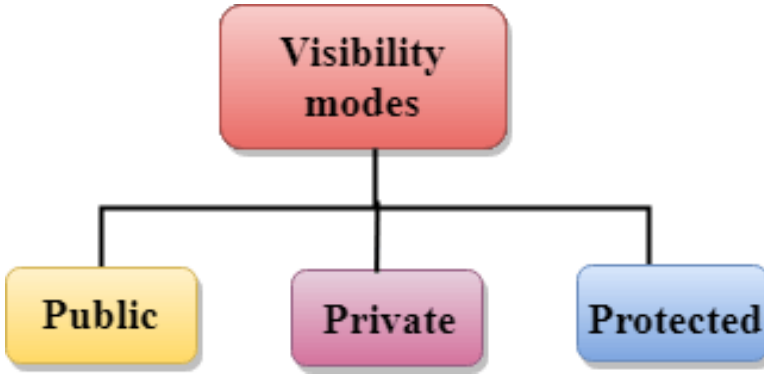
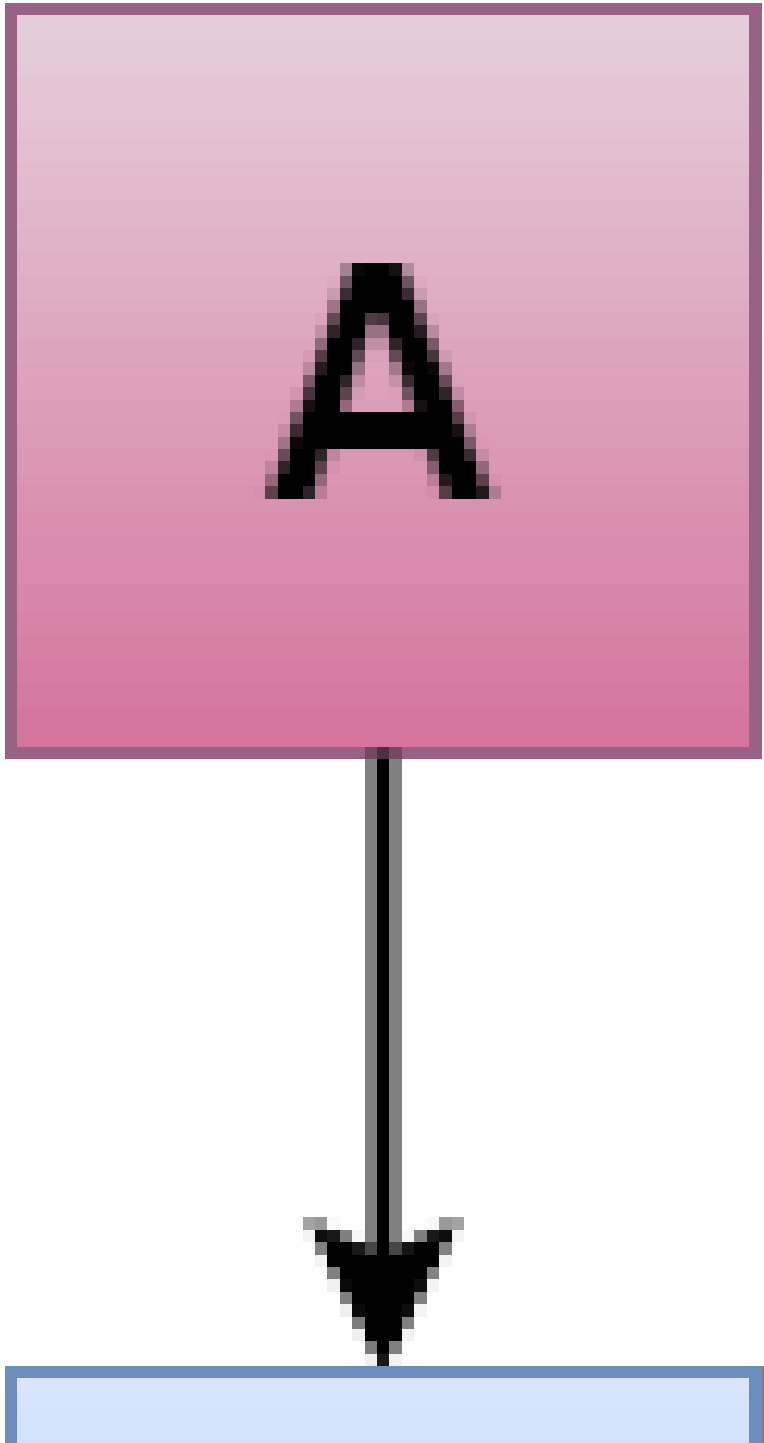


Figure 3.3: Visibiity modes


A

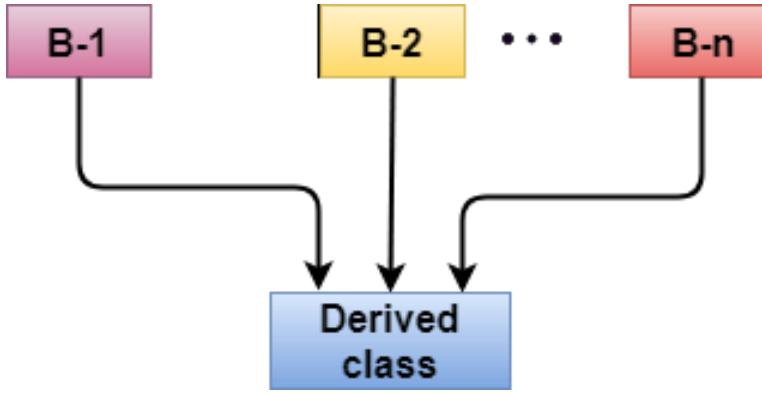


Figure 3.5: Multiple inheritance

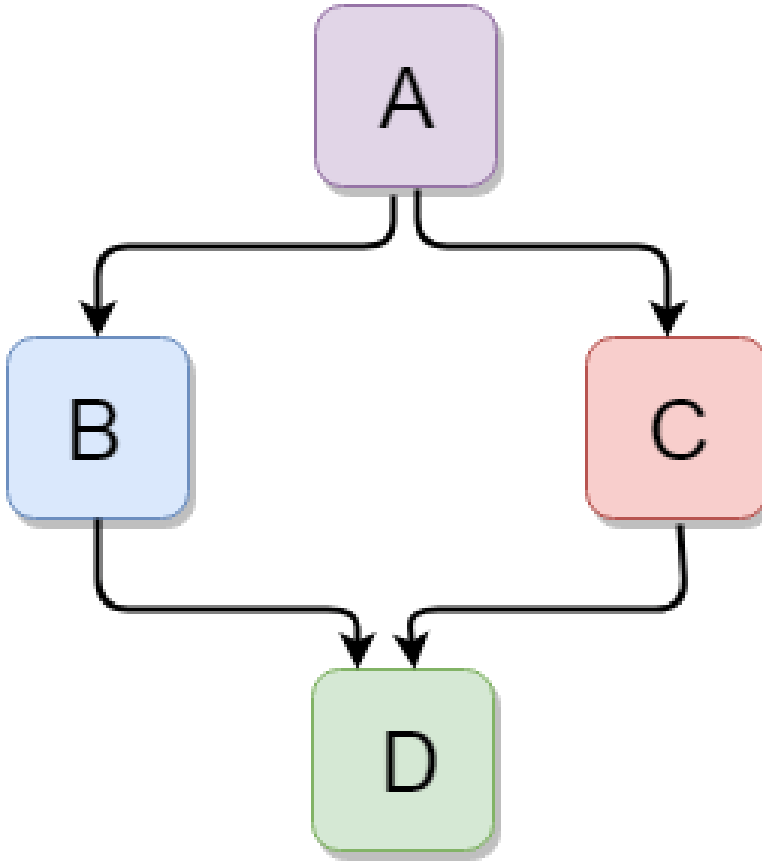


Figure 3.6: Hybrid inheritance

Figure 3.7: Hierarchical inheritance

Chapter 4

File Handling

4.1 What is file handling in C++?

Files store data permanently in a storage device. With file handling, the output from a program can be stored in a file. Various operations can be performed on the data while in the file.

A stream is an abstraction of a device where input/output operations are performed. You can represent a stream as either a destination or a source of characters of indefinite length. This will be determined by their usage. C++ provides you with a library that comes with methods for file handling. Let us discuss it.

4.2 The `fstream` Library

The fstream library provides C++ programmers with three classes for working with files. These classes include:

1. *ofstream*: This class represents an output stream. It's used for creating files and writing information to files.

2. *ifstream*: This class represents an input stream. It's used for reading information from data files.

Value	Description
ios::app	The Append mode. The output sent to the file is appended to it.
ios::ate	It opens the file for the output then moves the read and write control to file's end.
ios::in	It opens the file for a read.
ios::out	It opens the file for a write.
ios::trunc	If a file exists, the file elements should be truncated prior to its opening.

3. *fstream*: This class generally represents a file stream. It comes with ofstream/ifstream capabilities. This means it's capable of creating files, writing to files, reading from data files.

To use the above classes of the fstream library, you must include it in your program as a header file. Of course, you will use the `include` preprocessor directive. You must also include the `iostream` header file.

4.3 How to Open Files

Before performing any operation on a file, you must first open it. If you need to write to the file, open it using `fstream` or `ofstream` objects. If you only need to read from the file, open it using the `ifstream` object.

The three objects, that is, `fstream, ofstream`, and `ifstream`, have the `open()` function defined in them. The function takes this syntax:

```
1 open (file_name, mode);
```

1. The `file_name` parameter denotes the name of the file to open.
2. The mode parameter is optional. It can take any of the following values:

It is possible to use two modes at the same time. You combine them using the | *(OR)* operator.

4.3.1 Example 1

```
1 #include <iostream>
2 #include <fstream>
3 using namespace std;
```

```
int main() {
  fstream my_file;
  my_file.open("my_file", ios::out);
  if (!my_file) {
    cout << "File not created!";
  }
  else {
    cout << "File created successfully!";
    my_file.close();
  }
  return 0;
}
```

4.4 How to Close Files

Once a C++ program terminates, it automatically

1. flushes the streams
2. releases the allocated memory
3. closes opened files.

However, as a programmer, you should learn to close open files before the program terminates.

The `fstream, ofstream`, and `ifstream` objects have the `close()` function for closing files. The function takes this syntax:

```
void close();
```

4.5 Write to Files

You can write to file right from your C++ program. You use stream insertion operator `(<<)` for this. The text to be written to the file should be enclosed within double-quotes.

Let us demonstrate this.

4.5.1 Example 2

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

```
int main() {
  fstream my_file;
  my_file.open("my_file.txt", ios::out);
  if (!my_file) {
    cout << "File not created!";
  }
  else {
    cout << "File created successfully!";
    my_file << "Hello World!";
    my_file.close();
  }
  return 0;
}
```

4.6 Read from Files

You can read information from files into your C++ program. This is possible using stream extraction operator (>>). You use the operator in the same way you use it to read user input from the keyboard. However, instead of using the cin object, you use the ifstream/ fstream object.

4.6.1 Example 3

```
#include <iostream>
#include <fstream>
using namespace std;
int main() {
  fstream my_file;
  my_file.open("my_file.txt", ios::in);
  if (!my_file) {
    cout << "No such file";
  }
  else {
    char ch;

    while (1) {
      my_file >> ch;
      if (my_file.eof())
        break;

      cout << ch;
    }

  }
```

```
22   my_file.close();
23   return 0;
24 }
```

Chapter 5

Exception Handling

Exception Handling in C++ is a process to handle *runtime* errors. We perform exception handling so the normal flow of the application can be maintained even after *runtime* errors.

In C++, exception is an event or object which is thrown at runtime. All exceptions are derived from `std::exception` class. It is a runtime error which can be handled. If we don't handle the exception, it prints exception message and terminates the program. The main advantage is that it maintains the normal flow of the application. In such case, rest of the code is executed even after exception.

5.1 C++ Exception Classes

In C++ standard exceptions are defined in `<exception>` class that we can use inside our programs. The arrangement of parent-child class hierarchy is shown below:

5.1.1 Accessing Elements of Union

All the exception classes in C++ are derived from std::exception class. Let's see the list of C++ common exception classes.

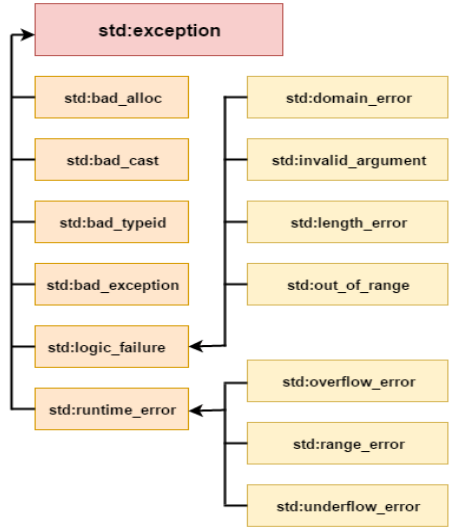


Figure 5.1: Errors and Exceptions

Exception	Description height
`std::exception`	It is an exception and parent class of all standard C++ exceptions.
`std::logic_failure`	It is an exception that can be detected by reading a code.
`std::runtime_error`	It is an exception that cannot be detected by reading a code.
`std::bad_exception`	It is used to handle the unexpected exceptions in a c++ program.
`std::bad_cast`	This exception is generally be thrown by *dynamic_cast*.
`std::bad_typeid`	This exception is generally be thrown by `typeid`.
`std::bad_alloc`	This exception is generally be thrown by new.

5.1.2 C++ Exception Handling Keywords

In C++, we use 3 keywords to perform exception handling:

1. try
2. catch, and
3. throw

Moreover, we can create user-defined exception which we will learn in next chapters.

5.2 C++ try/catch

In C++ programming, exception handling is performed using try/-catch statement. The C++ try block is used to place the code that may occur exception. The catch block is used to handle the exception.

5.2.1 C++ example without try/catch

```
#include <iostream>
using namespace std;
float division(int x, int y) {
   return (x/y);
}
int main () {
   int i = 50;
   int j = 0;
   float k = 0;
      k = division(i, j);
      cout << k << endl;
   return 0;
}
```

Output:

```
Floating point exception (core dumped)
```

C++ try/catch example

```
#include <iostream>
using namespace std;
float division(int x, int y) {
 if( y == 0 ) {
  throw "Attempted to divide by zero!";
 }
 return (x/y);
}
int main () {
   int i = 25;
   int j = 0;
   float k = 0;
   try {
      k = division(i, j);
      cout << k << endl;
   }catch (const char* e) {
      cerr << e << endl;
```

```
18    }
19    return 0;
20 }
```

Output:

```
Attempted to divide by zero!
```

5.3 C++ User-Defined Exceptions

The new exception can be defined by overriding and inheriting exception class functionality.

5.3.1 C++ user-defined exception example

Let's see the simple example of user-defined exception in which std::exception class is used to define the exception.

```
1  #include <iostream>
2  #include <exception>
3  using namespace std;
4  class MyException : public exception{
5    public:
6      const char * what() const throw()
7      {
8        return "Attempted to divide by zero!\n";
9      }
10 };
11 int main()
12 {
13   try
14   {
15     int x, y;
16     cout << "Enter the two numbers : \n";
17     cin >> x >> y;
18     if (y == 0)
19     {
20         MyException z;
21         throw z;
22     }
23     else
24     {
25         cout << "x / y = " << x/y << endl;
26     }
27   }
28   catch(exception& e)
```

```
  {
    cout << e.what();
  }
}
```

Output:

```
Enter the two numbers :
10
2
x / y = 5
```

Output:

```
Enter the two numbers :
10
0
Attempted to divide by zero!
```

5.4 Built-in Exceptions

5.4.1 bad_alloc exception

Standard C++ contains several built-in exception classes. The most commonly used is *bad_alloc*, which is thrown if an error occurs when attempting to allocate memory with new. This class is derived from exception. To make use of *bad_alloc*, one should set up the appropriate try and catch blocks. Here's a short example, that shows how it's used :

```
// CPP code for bad_alloc
#include <iostream>
#include <new>

// Driver code
int main () {
  try
  {
     int* gfg_array = new int[100000000];
  }
  catch (std::bad_alloc & ba)
  {
```

```
    std::cerr << "bad_alloc caught: " << ba.what();
  }
  return 0;
}
```

Output

```
RunTime error :
bad_alloc caught: std::bad_alloc
```

5.4.2 bad_cast exception

Standard C++ contains several built-in exception classes. `typeinfo::bad_cast` is one of them. This is an exception thrown on failure to dynamic cast. Below is the syntax for the same:

Header File:

```
<typeinfo>
```

Syntax:

```
class bad_cast;
```

To make use of `std::bad_cast`, one should set up the appropriate try and catch blocks. It doesn't return anything. Below are the examples to understand the implementation of std::bad$_c$*astinabetterway* :

Program 1:

```
// C++ code for std::bad_cast
#include <bits/stdc++.h>
#include <typeinfo>

using namespace std;

// Base Class
class Base {
    virtual void member() {}
};

// Derived Class
class Derived : Base {
};

// main() method
int main()
```

```
{

    // try block
    try {
        Base gfg;
        Derived& rd
            = dynamic_cast<Derived&>(gfg);
    }

    // catch block to handle the errors
    catch (bad_cast& bc) {
        cerr << "bad_cast caught: "
             << bc.what() << endl;
    }

    return 0;
}
```

Output:

```
bad_cast caught: std::bad_cast
```

Program 2:

```
// C++ code for std::bad_cast
#include <bits/stdc++.h>
#include <typeinfo>

using namespace std;

// Base Class
class Base {
    virtual void member() {}
};

// Derived Class
class Derived : Base {
};

// main() method
int main()
{

    // try block
    try {
        Base geeksforgeeks;
        Derived& abc
            = dynamic_cast<Derived&>(
                geeksforgeeks);
```

```
    }

    // catch block to handle the errors
    catch (bad_cast& a) {
        cerr << "bad_cast caught: "
             << a.what() << endl;
    }

    return 0;
}
```

Output:

```
bad_cast caught: std::bad_cast
```

5.4.3 bad_typeid exception

Standard C++ contains several built-in exception classes, `typeinfo::bad_typeid` is one of them. This is an exception thrown on typeid of null pointer. Below is the syntax for the same:

Header File:

```
<typeinfo>
```

Syntax:

```
class bad_typeid;
```

The `typeinfo::bad_typeid` returns a null terminated character that is used to identify the exception. To make use of `std::bad_typeid`, one should set up the appropriate try and catch blocks. Below are the examples to understand the implementation of `typeinfo::bad_typeid` in a better way:

Program 1:

```
// C++ code for std::bad_typeid
#include <bits/stdc++.h>

using namespace std;

struct gfg {
    virtual void func();
};

// main method
```

```
int main()
{
    gfg* g = nullptr;

    // try block
    try {
        cout << typeid(*g).name()
             << endl;
    }

    // catch block to handle the errors
    catch (const bad_typeid& fg) {
        cout << fg.what() << endl;
    }

    return 0;
}
```

Output:

```
std::bad_typeid
```

Program 2:

```
// C++ code for std::bad_typeid
#include <bits/stdc++.h>

using namespace std;

struct geeksforgeeks {
    virtual void
    A_Computer_Science_Portal_For_Geeks();
};

// main method
int main()
{
    geeksforgeeks* gfg = nullptr;

    // try block
    try {
        cout << typeid(*gfg).name() << endl;
    }

    // catch block to handle the errors
    catch (const bad_typeid& fg) {
        cout << fg.what() << endl;
    }

```

```
    return 0;
}
```

Output:

```
std::bad_typeid
```

5.4.4 bad_exception exception

Standard C++ contains several built-in exception classes, `exception::bad_except` is one of them. This is an exception thrown by unexpected handler.
Below is the syntax for the same:

Header File:

```
include<exception>
```

Syntax:

```
class bad_exception;
```

The `exception::bad_exception` returns a null terminated character that is used to identify the exception. To make use of `exception::bad_exceptio` one should set up the appropriate try and catch blocks. Below are the examples to understand the implementation of `exception::bad_exception` in a better way:

Program 1 :

```
// C++ code for std::bad_exception
#include <bits/stdc++.h>

using namespace std;

void func()
{
    throw;
}

void geeksforgeeks() throw(bad_exception)
{
    throw runtime_error("test");
}

// main method
int main()
{
```

```
    set_unexpected(func);

    // try block
    try {
        geeksforgeeks();
    }

    // catch block to handle the errors
    catch (const bad_exception& gfg) {
        cout << "Caught exception "
             << gfg.what() << endl;
    }
    return 0;
}
```

Output:

```
Caught exception std::bad_exception
```

Program 2 :

```
// C++ code for std::bad_exception
#include <bits/stdc++.h>

using namespace std;

void gfg()
{
    throw;
}

void A_Computer_Science_Portal_For_Geeks()
     throw(bad_exception)
{
    throw runtime_error("test");
}

// main method
int main()
{
    set_unexpected(gfg);

    // try block
    try {
        A_Computer_Science_Portal_For_Geeks();
    }

    // catch block to handle the errors
    catch (const bad_exception& a) {
```

```
        cout << "Caught exception "
             << a.what() << endl;
    }
    return 0;
}
```

Output:

```
Caught exception std::bad_exception
```

5.4.5 `logic_error` exception

The class serves as the base class for all exceptions thrown to report errors presumably detectable before the program executes, such as violations of logical preconditions.

Syntax

```
class logic_error : public exception {
public:
    explicit logic_error(const string& message);

    explicit logic_error(const char *message);

};
```

Remarks

The value returned by what() is a copy of message.data(). For more information, see what and data.

Example

```
// logic_error.cpp
// compile with: /EHsc
#include <exception>
#include <iostream>
#include <stdexcept>
#include <typeinfo>
using namespace std;

int main()
{
   try
   {
      throw logic_error("Does not compute!");
```

```
    }
    catch (const exception& e)
    {
        cerr << "Caught: " << e.what() << endl;
        cerr << "Type: " << typeid(e).name() << endl;
    }
}
/* Output:
Caught: Does not compute!
Type: class std::logic_error
*/
```

5.5 Runtime error

In this article, we will discuss the reason for the run-time error and its solution. A *runtime error* in a program is an error that occurs while the program is running after being successfully compiled. Below are some methods to identify the reason behind Runtime error:

5.5.1 Invalid memory access

When the index of the array is assigned with a negative index it leads to invalid memory access during runtime error. Below is the C++ Program to illustrate the invalid memory access during run-time:

```
// C++ program to illustrate invalid
// memory access during run-time
#include <iostream>
using namespace std;

// Global declaration
int arr[5];

// Driver Code
int main()
{
    int answer = arr[-10];
    cout << answer;

    return 0;
}
```

Output:

```
1736487104
```

5.5.2 Array runs out of bound

Sometimes Array or vector runs out of bound of their limits resulting in a runtime error. Below is the C++ program illustrating array runs out of bound:

```
// C++ program to illustrate
// array runs out of bound
#include <iostream>
using namespace std;

// Driver Code
int main()
{
    long n;
    n = 100000000000;

    // 'n' is out of bound for
    // the array limit
    long a[n];

    cout << a[1] << " ";
    return 0;
}
```

It can be resolved by using the size of the array/vector as within the limit.

5.5.3 Unassigned variables

Some silly mistakes encountered while coding in hurry, sometimes leads to runtime error. Below is the C++ program illustrating runtime error by un-assigned variables:

```
// C++ program to illustrate runtime
// error by un-assigned variables
#include <iostream>
using namespace std;

// Driver Code
int main()
{
```

```
    long long N;

    // N is assigned garbage value
    long arr[N];

    cin >> N;

    for (int i = 0; i < N; i++) {
        cin >> arr[i];
    }

    for (int i = 0; i < N; i++) {
        cout << arr[i] << " ";
    }

    return 0;
}
```

Here, variable N is assigned a garbage value resulting in a runtime error. Sometimes, since it depends on the compiler how it assigned the garbage value. This can be resolved by declaring *arr[N]* after scanning the value for variable n and checking if it is the upper or lower limit of the array/vector index.