C++

Course With Harry

INDEX

53. this Pointer in C++ | 53

54. Polymorphism in C++ | 54

55. Pointers to Derived Classes in C++ | 5

56. Virtual Functions in C++ | 56

57. Virtual Functions Example + Creation Rules in C++ | 57

58. Abstract Base Class & Pure Virtual Functions in C++ | 58

59. File I/O in C++: Working with Files | 59

60. File I/O in C++: Reading and Writing Files | 60

61. File I/O in C++: Read/Write in the Same Program & Closing Files | 61

62. File I/O in C++: open() and eof() functions | 62

63. C++ Templates: Must for Competitive Programming | 63

64. Writing our First C++ Template in VS Code | 64

65. C++ Templates: Templates with Multiple Parameters | 65

66. C++ Templates: Class Templates with Default Parameters | 66

67. C++ Function Templates & Function Templates with Parameters | 67

68. Member Function Templates & Overloading Template Functions

69. The C++ Standard Template Library (STL) |

70. Containers in C++ STL | 70

71. Vector In C++ STL | 71

72. List In C++ STL | 72

73. Map In C++ STL | 73

74. Function Objects (Functors) In C++ STL |

Introduction to C++, Installing VS Code, g++ & more | 1

In these C++ tutorials, we are going to learn about the C++ programming language from the very basics to the industry level. This course has been designed in a way to nurture the beginners setting their feet in this discipline. So tighten your belts, and enjoy the ride!

Before starting, let me explain to you the difference between this and any other source for learning C++:

1. This is a whole complete package in itself, covering from A to Z of the language.
2. No redundancy and monotonicity.

3. Beginner friendly, hence easy to learn.
4. No prerequisite, just a zeal for learning.

To start with, I'll be covering these things in today's tutorial:

- What is programming and why C++?
- Installation of Visual Studio Code
- Installation of g++
- Writing our first program and executing it.

What is programming, programming language, and why C++?

Programming can be understood as your instructions to the computer(machine) to solve real problems. The very basic principle of creating machines was to make life simpler and machines, on our instructions, have been able to do the same. But the distance between what we say and what the machine understands gets abridged by a programming language. This marks the importance of learning a programming language.

A programming language helps us communicate with the computer. Analogous to us humans, who need some language, be it English, Hindi, or Bangla, to talk to our people, computers too need a language to converse. Just to name a few, there is C++, C, Python, Java, etc.

Now when we start listing the names of these programming languages, the question which instinctively arises is why C++. Despite this being an 80s programming language, it never lost its sheen. C++ was an added version of C developed by Bjarne Stroustrup. It is believed to be very close to the hardware making it comparatively easy for programmers to give the instructions directly to the system without any intermediary. Another reason for C++ to be one of the most used languages is its object-orientedness. C++ is an object-oriented programming language giving it the power to create real-world software systems. You don't have to worry much about these terms, which believe me, only sound complex and ain't really! Just sit back and dive with me in this plethora of knowledge.

Installation of Visual Studio Code

Visual studio is a source code editor - free to use, provided & maintained by Microsoft. Below is the process of downloading and installing visual studio code:

Step 1: Click here and you will be redirected to the official download page of VS Code. Download the VS code according to your operating system in use. I will be downloading it for Windows 10 as shown in the below animation.

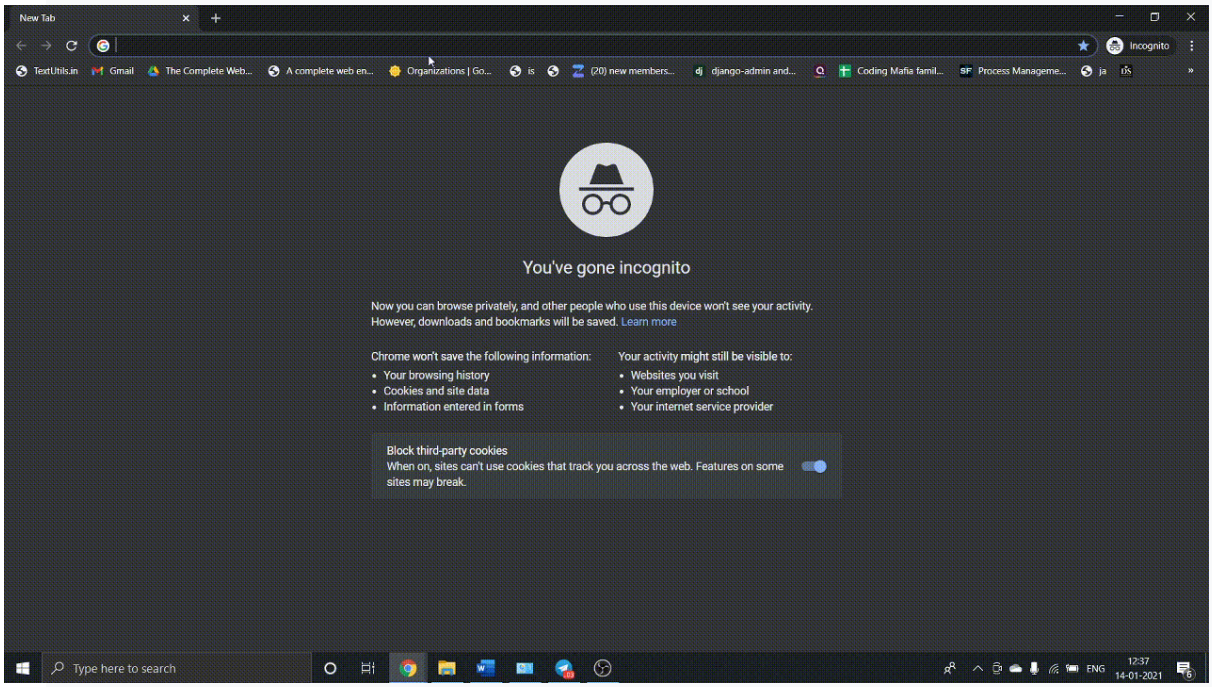


Figure 1: Visual Studio Code Website

Step 2: Once the downloading is complete, install VS code on your system like any other application.

Installation of g++

g++ is a compiler that helps us convert our source code into a .exe file. Below is the process of downloading and installing g++:

Step 1: Go to Google and search "MinGW install" and click on the MinGW link, as shown in the image below.

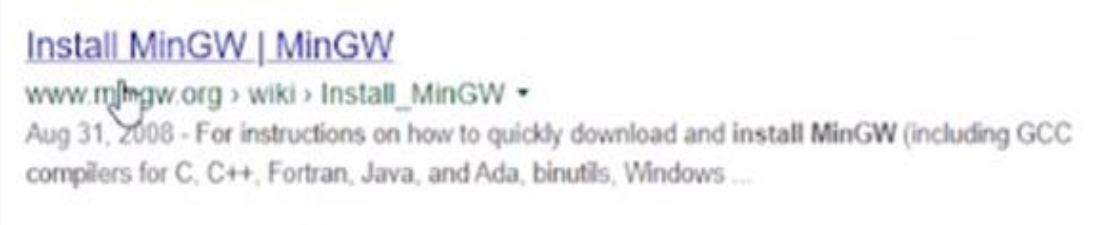


Figure 4: g++ installation from Google

Step 2: Click on the download button on the top right corner menu.

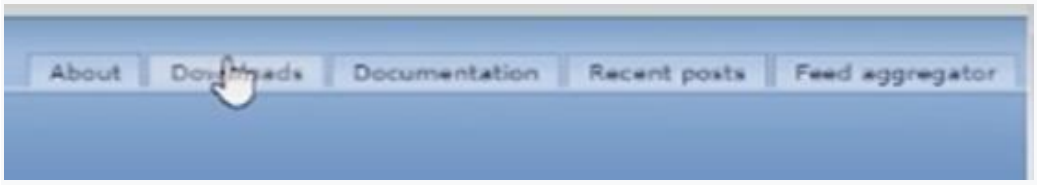


Figure 5: G++ Download Step

Step 3: After visiting the download page, click on the windows button as shown in the image below to start the downloading

Figure 6 G++ Download Step

Step 4: After the download - open the program and click "Continue" to start the installation process.

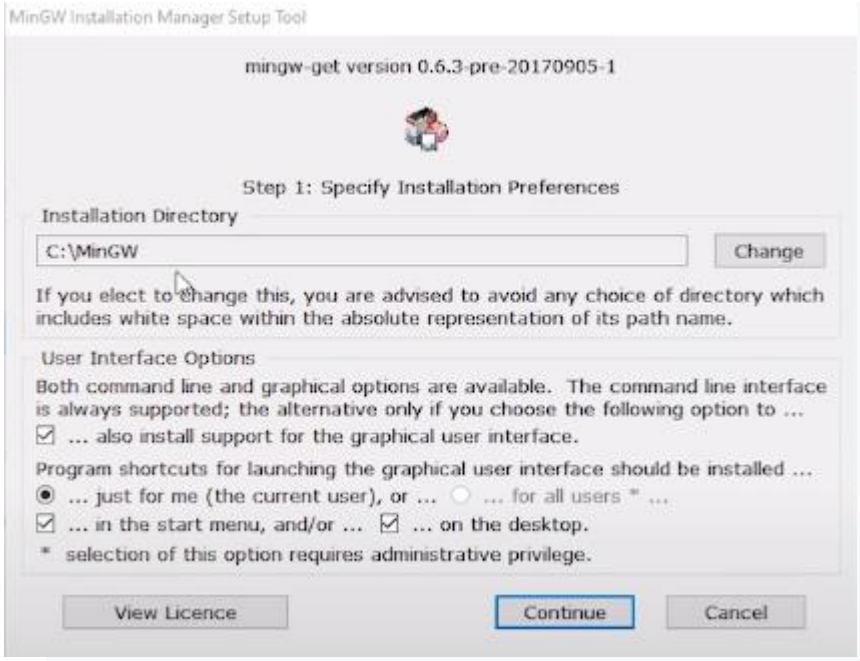


Figure 7: G++ Installation Step

Step 5: After downloading some packages, it will show you a screen, as shown in the image below. You have to mark both the boxes as in the image below, and then click on installation on the top left corner menu. Finally, click apply changes, and it will start downloading the required packages.

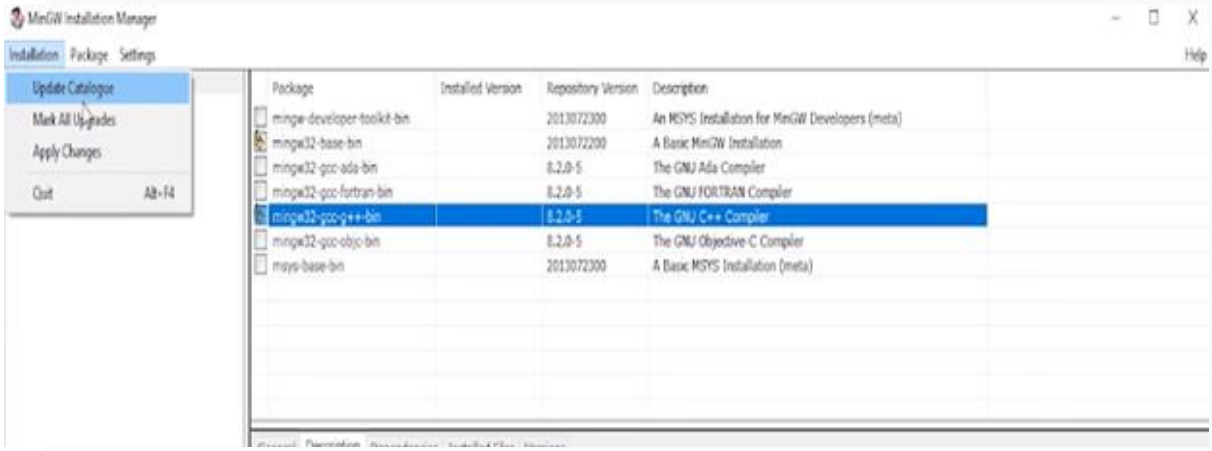


Figure 8: G++ Installation Step

Step 6: After finishing step 5, close the program and open C:// drive. Furthermore, locate the MinGW folder. Go to its bin directory and copy its file path, as shown in the image below.

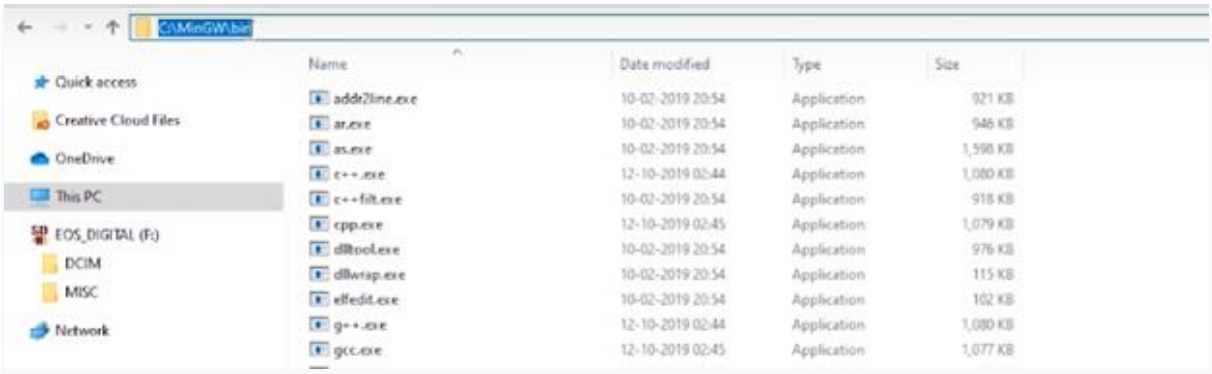


Figure 9: Copying g++ file path

Step 7: Now right-click on this pc and go to properties

Step 8: After that click on advanced system settings as shown in the image below

Figure 10: Step to Add g++ File Path

Step 9: After that click on "Environment Variables" as shown in the image below

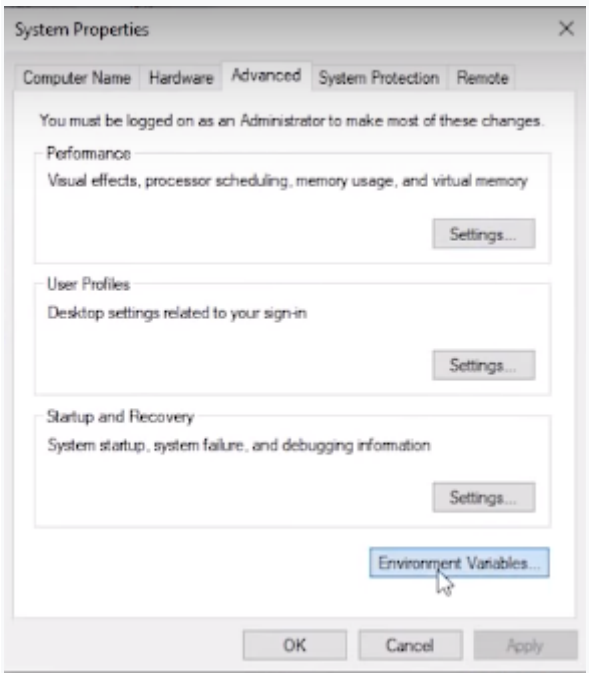


Figure 11: Step to Add G++ File Path

Step 10: After that select path and click on edit as shown in the image below

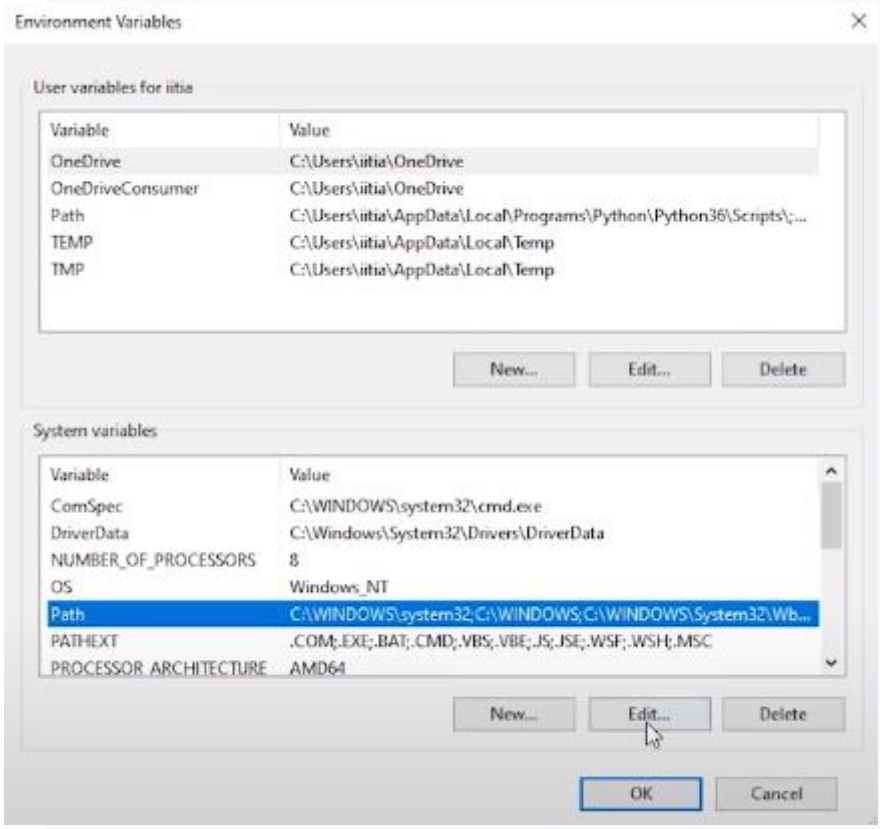


Figure 12: Step to Add g++ file path

Step 11: Then, click on new and paste the file path and click ok as shown in the image below

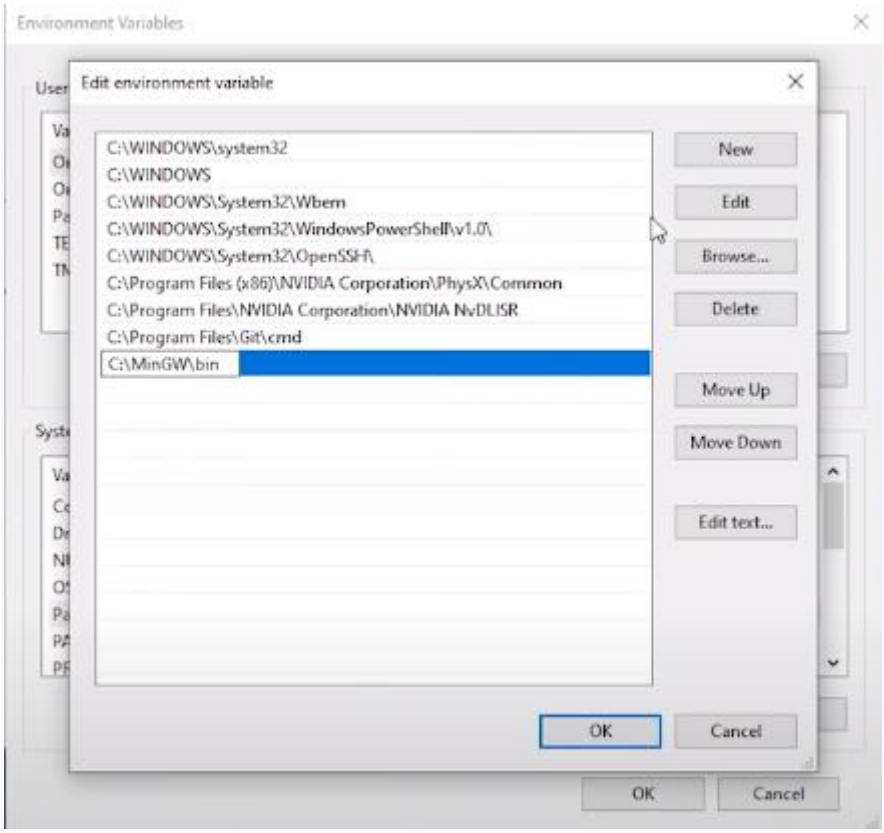


Figure 13: Adding g++ File Path

After adding the file path now, our g++ compiler is ready, and we can start coding now.

Writing Our First Program and Executing It

To write our first program, we need Visual studio code, which is a source code editor. Create a folder and then right-click inside the folder and click on "**open with code"** as shown below:

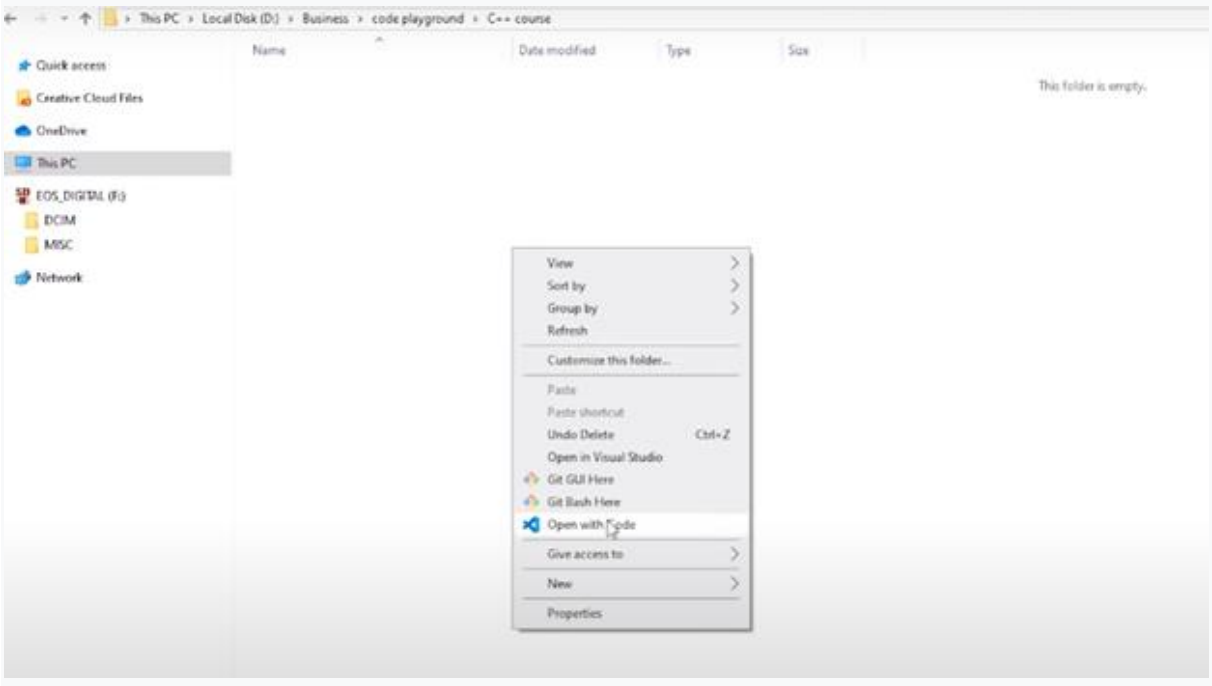


Figure 14: Opening Visual Studio Code in a folder

Following these steps will open your visual studio code with that folder as the context. After opening the VS Code, you have to install some extensions. Go to the extension menu and search "**C/C++,"** and it will show you this extension. C/C++, as well as the other extensions we add, will make our life easy while learning C++. Click on the install button, and it will start installing the extension for you, as shown in the image below.

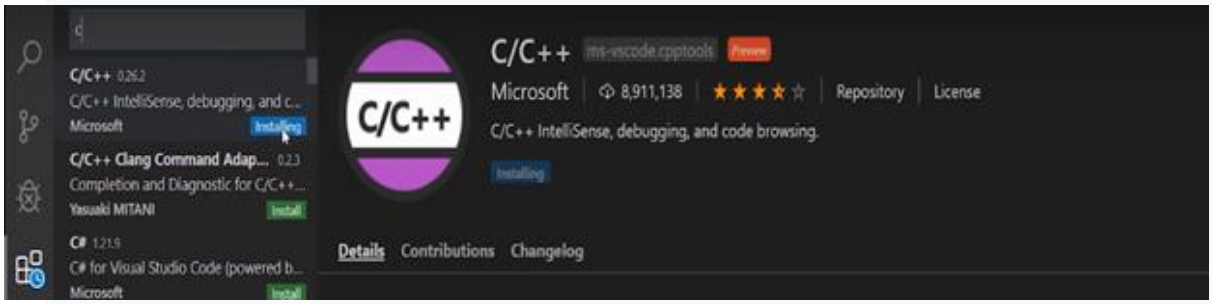


Figure 15: Installing Extension

This extension will help us in writing code through features such as auto-complete or auto-dropdown suggestions. Let us install one more extension, which will help us run our programs quickly. Go to the Extensions tab in the top left corner and search "**Code runner."** After that, click on install.

Figure 16: Installing Extension

Now we have to create our program file and start writing our code. To create a program file, you have to go to the File menu > then click on the file button, as shown in the image below.

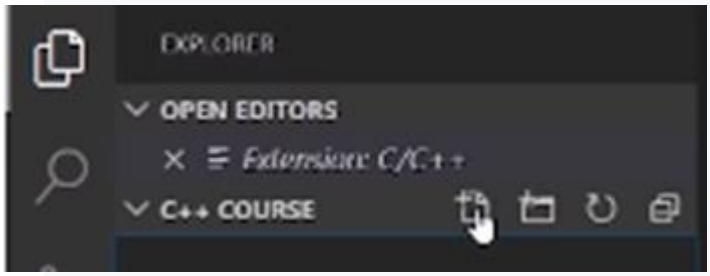


Figure 17: Creating a Program File

After clicking on the file button, it will ask you for the file name. Give the name of the file as "tutorial1.cpp" and press enter. Now the code file will be created, and you can start writing your program.

Figure 18: Running Code

In today's tutorial, we are not going to learn anything about what this code is all about. We will learn these things step by step in our upcoming lectures. Now to execute this code, press the run button, as shown in the image above, and it will give you the output as shown in the image below.

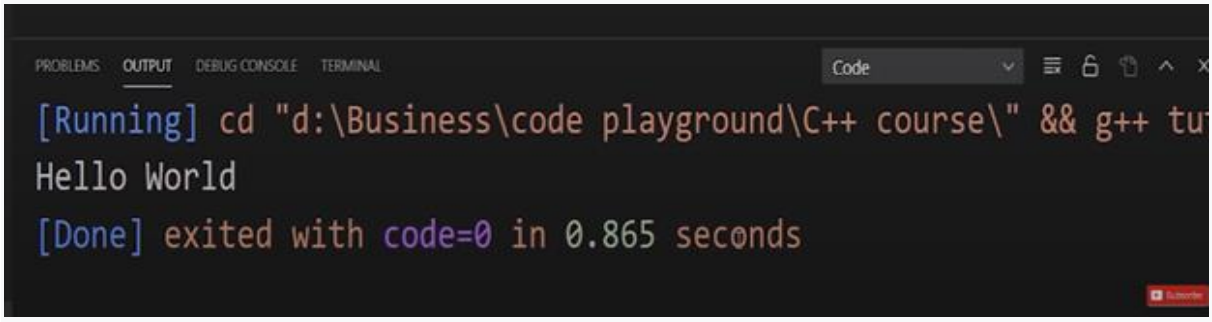


Figure 19: Program Output

Thank you friends for starting to learn C++ with me, hope you liked the tutorial. If you haven't checked out the whole playlist yet, move on to codewithharry.com or my YouTube channel to access it. I hope you enjoy them.

In the next tutorial, we'll be talking more about the basic structures of a C++ program, see you there, till then keep coding.

Code as described/written in the video

```
include<iostream>


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

Basic Structure of a C++ Program | 2

In this series of C++ tutorials, we will visualize the basic structure of a C++ program. In our last lesson, we discussed C++, and we had also learned about how to download and install visual studio code and g++; If you haven't read the previous lecture, feel free to navigate through the course content!

In our previous lecture, we had written and executed a small C++ program. Today we are going to discuss that program in more detail.

```
1  #include<iostream>
2
3  int main(){
4      std::cout<<"Hello World";
5      return 0;
6  }
```

Figure 1: C++ Code from the previous tutorial

As you can see in the image, this was the program that we had executed in our previous lecture. Now we will discuss what each line of code in the program does.

1. Let's start with the 1st line of code "**include<iostream>"** - this whole text is called a **header file**.

In this line of code **include** is a keyword used to add libraries in our program. "**iostream"** is the name of a library, added to our program. The **iostream** library helps us to get input data and show output data. The iostream library also has many more uses; it is not only limited to input and output but for now, we will not go into more detail on this.

2. One more thing to consider here is that the 2nd line of code is blank; there is no code written. The thing to consider here is that it doesn't matter how many lines you have left empty in a C++ program, the compiler will ignore those lines and only check the ones with the code.

3. Now onto the 3rd line of code "**int main() {" -** In this line of code, "**int"** is a return type which is called integer and "**main()"** is a function, the brackets "**()"** denotes that it is a function. The curly brace "**{"** indicates that it is an opening of a function, and the curly brace "**}"** indicates that it is the closing of a function. Here I will give you **an example** to better understand the functionality of "**int main()."** Imagine that you own a coffee shop, and you have some employees who work for you. Let's name **(Anna, Blake, Charlie)** as the three employees. The function of Anna is to take orders, the function of Blake is to make coffee, and Charlie's function is to deliver coffee. Now when Anna gets a coffee order, she will call Blake to make the coffee, and when the coffee is ready, Blake will call Charlie to deliver the coffee. In this scenario, we can say that **Anna is the primary function** from which all the other tasks will start, and **coffee is our return value (Something charlie finally gives to Blake)**. In this line of code, "**main"** is a reserved keyword, and we cannot change it to some other keyword. This was just an analogy, and you will understand this very well in upcoming tutorials.

4. Now let's talk about the 4th line of code '**std::cout<<" hello world";' -** In this line of code "**std"** is a namespace, "::" is the scope resolution operator and "**cout<<"** is a function which is used to output data, "**hello world"** is our output string and "**;"** is used to end a statement. The code "**std::cout"** tells the compiler that the "cout" identifier resides in the std namespace. Main key points here are:

- We can write as many statements as you want in a single line, but I recommend you write one statement per line to keep the code neat and clean.
- Anything which is written between double quotation " " is a string literal (More on strings later).

5. Now let's talk about the 5th line of code "**return 0"** - In this line of code, the return keyword will return 0 as an integer to our main function "**int main()"** as we have discussed before. Returning 0 as a value to the main function means successful termination of the program.

So that was the anatomy of a C++ program. I hope you understood the functions of various parts in a C++ program.

In this tutorial, we will learn about the variables and comments in the C++ language. In our last lesson, we discussed the basic structure of a C++ program, where we understood the working of the C++ code line by line. If you haven't read the previous lecture, make sure to navigate through the course content section.

We are going to cover two important concepts of C++ language:

- **Variables in C++**
- **Comments in C++**

Before explaining the concept of variables and comments, I would like to clarify two more ideas**: low level** and **high level.** To make it easy to understand, let's consider this scenario - when we go to Google search engine and search for some queries, Google displays us some websites according to our question. Google does this for us at a very high level. We don't know what's happening at the low level until we look into Google servers (at a low level) and further to the level where the data is in the form of 0s/1s. The point I want to make here is that low level means nearest to the hardware, and a high level means farther from the hardware with a lot of layers of abstraction.

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

// This program was created by Code With Harry
/* this
is
a
multi
line
comment */
int main(){
    int sum = 6;
    cout<< "Hello world"<< sum;
    return 0;
}
```

Figure 1: C++ Sample Code

Variables in C++

Variables are containers to store our data. To make it easy to understand, I will give a scenario: to store water, we use bottles, and to store sugar, we use a box. In this scenario, the bottle and box are containers that are storing water and sugar; the same is the case with variables; they are the containers for data types. As there are many types of data types, for example, "**int"** is used for integers, the "**float"** is used for floating-point numbers, "**char"** is used for character, and many more data types are available, we will discuss them in upcoming lectures. The main point here is that these variables store the values of these data types. Figure 1 shows an example of a variable. "**sum"** is taken as an integer variable, which will store a value 6, and writing sum after the "**cout"** statement will show us the value of "sum" on the output window.

Comments in C++

A comment is a human-readable text in the source code, which is ignored by the compiler. There are two ways to write comments.

Single-Line Comments: 1st way is to use" //" before a single line of text to make it unparsable by the compiler.

Multi-Line Comments: 2nd way is to use "/*" as the opening and "*/" as the closing of the comment. We then write text in between them. If we write text without this, the compiler will try to read the text and will end up giving an error. Figure 1 shows examples of these comments.

Code as described/written in the video

```
include<iostream>
using namespace std;


// This program was created by Code With Harry
```

```
/* this
is
a
multi
line
comment */
int main(){
    int sum = 6;
    cout<< "Hello world"<< sum;
    return 0;
}
```

Variable Scope & Data Types in C++ in Hindi | 4

In this series of C++ tutorials, we will visualize the variable scope and data types in the C++ language in this lecture. In our last lesson, we discussed the variable's role and comments. In this C++ tutorial, we are going to cover two important concepts of C++:

- **Variable Scope**
- **Data Types**

Before explaining the concept of variable scope, I would like to clarify about variables a little more. Variable can be defined as a container to hold data. Variables are of different types, for example:

1. **Int**-> Int is used to store integer data e.g (-1, 2, 5,-9, 3, 100).
2. **Float**-> Float is used to store decimal numbers e.g (0.5, 1.05, 3.5, 10.5)
3. **Char**-> Char is used to store a single character e.g. ('a', 'b',' c', 'd')
4. **Boolean**-> Boolean is used to store "true" or "false."
5. **Double**-> Double is also used to store decimal numbers but has more precision than float, e.g. (10.5895758440339...)

Here is an example to understand variables: int sum = 34; means that sum is an integer variable that holds value '34' in memory.

Syntax for Declaring Variables in C++

1. Data_type Variable_name = Value;
2. Ex: int a=4; char letter = ‘p’;
3. Ex: int a=4, b=6;

Variable Scope

The scope of a variable is the region in the program where the existence of that variable is valid. For example, consider this analogy - if one person travels to another country illegally, we will not consider that country as its scope because he doesn't have the necessary documents to stay in that country.

Base on scope, variables can be classified into two types:

- **Local variables**
- **Global variables**

Local variables:

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

Global variables:

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

Data Types

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

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

- **Built-in**
- **User-defined**
- **Derived**

1. Built-in Data Types in C++:

- Int
- Float
- Char
- Double
- Boolean

2. User-Defined Data Types in C++:

- Struct
- Union
- Enum

Note: We will discuss the concepts of user-defined data types in another lecture. For now, understanding that these are user-defined data types is enough.

3. Derived Data Types in C++:

- Array
- Pointer
- Function

Note: We will discuss the concept of derived data types in another lecture. For now, understanding that these are derived data types is enough.

Practical Explanation of Initializing Variables

We have discussed a lot in theory now; we will see the actual code and its working. The code for initializing variables is shown in Figure 1.

```
5  int main(){
6      // int a = 14;
7      // int b = 15;
8      int a=14, b=15;
9      float pi=3.14;
10     char c ='d';
11     cout<<"This is tutorial 4.\nHere the value of a is "<<a<<".\nThe
       value of b is "<< b;
12     cout<<"\nThe value of pi is: "<<pi;
13     cout<<"\nThe value of c is: "<<c;
14     return 0;
15 }
```

Figure 1: initializing variables and printing their values

In this code, we have initialized different types of variables and then printed them on screen. At line no 6 and 7, we initialized two integer variables a, and b, but they are commented out for compiler to ignore them. At line no 8, we have again initialized two integer variables a, and b, but this time they both are on the same line separated by a comma. The main thing to note here is that variables can be initialized on separate lines or in a single line. At line no 9, we have initialized a float variable pi and assigned a decimal value 3.14 to it. At line no 10, we have initialized a character variable

c and assigned a character'd' to it. At line no 11, we have printed the value of a, and b. The main thing to note here is that "**/n"** is used to print a new line. At line no 12, we have printed the value of pi. Similarly, in line no 13, the value of c is printed. The output for this program is shown in figure 2.

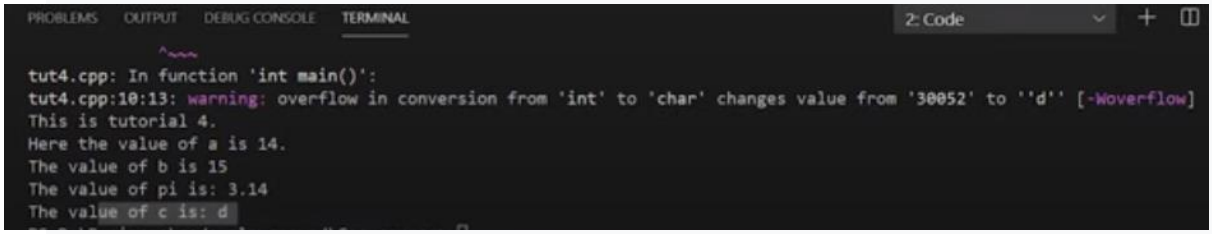


Figure 2: Output of the Program

Practical Explanation of Variables Scope

We have discussed the variable scope in theory now; we will see its actual code and working. So the code for variables scope is shown in figure 3.

```
# include<iostream>

using namespace std;
int glo = 6;
void sum(){
    int a;
    cout<< glo;
    }

int main(){
    int glo=9;
    glo=78;
    // int a = 14;
    // int b = 15;
    int a=14, b=15;
    float pi=3.14;
    char c ='d';
    bool is_true = false;
    sum();
    cout<<glo<< is_true;
    // cout<<"This is tutorial 4.\nHere the value of a is "<<a<<".\nThe value of b is "<< b;
    // cout<<"\nThe value of pi is: "<<pi;
    // cout<<"\nThe value of c is: "<<c;
    return 0;
}
```

Figure 3: Variables Scope Code

In this piece of code, we have initialized two "glo" variables. 1st variable is outside the main function, and the 2nd variable is inside the main function. The value assigned to the "glo" variable outside the main function is 6, and the value assigned to the "glo" variable inside the main function is 9. One thing to note here is that in the main function, we have again assigned a value 78 to the variable "glo" which will update the previous value 9.

After initializing the "glo" variables, we had output the "glo" variables at two places in our program the 1st place is inside the main function, and the 2nd place is inside the sum function. **The main thing to note here is that:**

- When the "cout" will run inside the sum function, it will check for "glo" variable value inside the sum function. As we can see that there is no "log" variable initialized inside the sum function, it will check for the "glo" variable outside of the sum function scope, which we call a global scope. As we can see, that "glo" function is initialized in the global scope with the value 6, so the sum function will take that value.
- When the "cout" will run inside the main function, it will check for "glo" variable value inside the main function first, and as we can see that there is a "glo" variable initialized inside the main function scope which is a local scope, it will use that value.

The output of this program is shown in figure 4.

Figure 4: Variable Scope Output

As we can see that the output is "6781". The output 6 is from the cout of "glo" in sum function, the output 78 is from the cout of "glo" in the main function, and the output 1 is from the cout of "is_true" Boolean variable in the main function.

Rules to Declare Variables in C++

- Variable names in C++ language can range from 1 to 255
- Variable names must start with a letter of the alphabet or an underscore
- After the first letter, variable names can contain letters and numbers
- Variable names are case sensitive
- No spaces and special characters are allowed
- We cannot use reserved keywords as a variable name

Code as described/written in the video

```
 include<iostream>


using namespace std;
int glo = 6;
void sum(){
    int a;
    cout<< glo;
    }

int main(){
    int glo=9;
    glo=78;
    // int a = 14;
    // int b = 15;
    int a=14, b=15;
    float pi=3.14;
    char c ='d';
    bool is_true = false;
    sum();
    cout<<glo<< is_true;
    // cout<<"This is tutorial 4.\nHere the value of a is "<<a<<".\nThe value of b is "<< b;
    // cout<<"\nThe value of pi is: "<<pi;
    // cout<<"\nThe value of c is: "<<c;
    return 0;
```

In this tutorial, we will visualize basic input and output in the C++ language. In our last lesson, we discussed the variable's scope and data types. In this C++ tutorial, we are going to cover basic input and output:

Basic Input and Output in C++

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

Input stream

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

Output stream

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

Practical Explanation of Input/Output

We will see the actual code for input/output, and it's working. Consider the code below:

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

int main()
{
    int num1, num2;
    cout<<"Enter the value of num1:\n"; /* '<<' is called Insertion operator */
    cin>>num1; /* '>>' is called Extraction operator */

    cout<<"Enter the value of num2:\n"; /* '<<' is called Insertion operator */
    cin>>num2; /* '>>' is called Extraction operator */

    cout<<"The sum is "<< num1+num2;

    return 0;
}
```

Figure 1: Basic input/output program

In this piece of code, we have declared two integer variables "**num1"** and "**num2"**. Firstly we used "**cout"** to print "**Enter the value of num1**:" as it is on the screen, and then we used "**cin"** to take the input in "**num1"** at run time from the user.

Secondly, we used "**cout"** to print "**Enter the value of num2**:" as it is on the screen, and then we used "**cin"** to take the input in "**num2"** at run time from the user.

In the end, we used "**cout"** to print "**The sum is"** as it is on the screen and also gave the literal "**num1+num2"** which will add the values of both variables and print it on the screen.

The output of the following program is shown in figure 2.

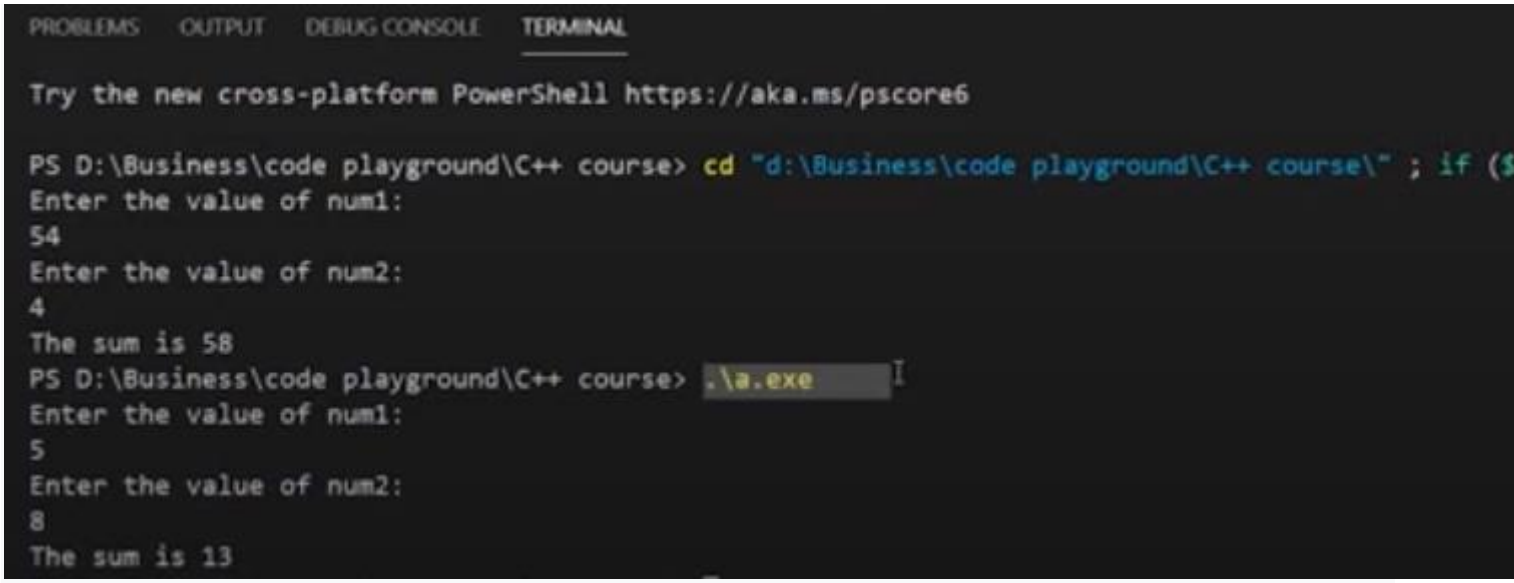


Figure 2: Output of the Program

We have executed our program two times, which can be seen in figure 2. In our 1st execution, we had input the value "**54"** for the variable "**num1"** and value "**4"** for the variable "**num2"**. This gives us the sum of both numbers as "**58"**.

In our 2nd execution, we had input the value "**5"** for the variable "**num1"** and value "**8"** for the variable "**num2"**. This gives us the sum of both numbers as "**13"**.

Important Points

1. The sign "**<<"** is called insertion operator
2. The sign "**>>"** is called extraction operator
3. "**cout"** keyword is used to print
4. "**cin"** keyword is used to take input at run time.

Reserved keywords in C++

Reserved keywords are those keywords that are used by the language itself, which is why these keywords are not available for re-definition or overloading. In short, you cannot create variables with these names. A list of reserved keywords is shown in figure 3.

Figure 3: Reserved keywords in C++

Code as described/written in the video

```
 include<iostream>
using namespace std;

int main()
{
    int num1, num2;
    cout<<"Enter the value of num1:\n"; /* '<<' is called Insertion operator */
    cin>>num1; /* '>>' is called Extraction operator */

    cout<<"Enter the value of num2:\n"; /* '<<' is called Insertion operator */
    cin>>num2; /* '>>' is called Extraction operator */

    cout<<"The sum is "<< num1+num2;

    return 0;}
```

In this C++ tutorial, we will talk about header files and operators. In our last lesson, we discussed the basic input and output. Lets now cover header files and operators in C++ language:

Header Files in C++

"**include"** is used in C++ to import header files in our C++ program. The reason to introduce the **"**<**iostream**>" header file into our program is that functions like "**cout"** and "**cin"** are defined in that header file. There are two types of header files:

System Header Files

System header files ships with the compiler. For example, "**include <iostream>**". To see the references for header files click here

User-Defined Header Files

The programmer writes User-defined header files himself. To include your header file in the program, you first need to make a header file in the current directory, and then you can add it.

Operators in C++

Operators are used for producing output by performing various types of calculations on two or more inputs. In this lecture, we will see the operators in C++.

Arithmetic Operators

Arithmetic operators are used for performing mathematical operations like (+, -, *). The arithmetic operators are shown in Figure 1.

```
int a=4, b=5;
cout<<"Operators in C++:"<<endl;
cout<<"Following are the types of operators in C++"<<endl;
// Arithmetic operators
cout<<"The value of a + b is "<<a+b<<endl;
cout<<"The value of a - b is "<<a-b<<endl;
cout<<"The value of a * b is "<<a*b<<endl;
cout<<"The value of a / b is "<<a/b<<endl;
cout<<"The value of a % b is "<<a%b<<endl;
cout<<"The value of a++ is "<<a++<<endl;
cout<<"The value of a-- is "<<a--<<endl;
cout<<"The value of ++a is "<<++a<<endl;
cout<<"The value of --a is "<<--a<<endl;
cout<<endl;
```

Figure 1: Arithmetic Operators

1. The function "**a+b"**, will add a and b values and print them.
2. The function "**a-b "**will subtract a and b values and print them.
3. The function "**a*b"** will multiply a and b values and print them.
4. The function "**a/b "**, will divide a and b values and print them.
5. The function "**a%b "**, will take the modulus of a and b values and print them.
6. The function "**a++"** will first print the value of a and then increment it by 1.
7. The function "**a--"**, will first print the value of a and then decrement it by 1.
8. The function "**++a"**, will first increment it by one and then print its value.
9. The function "**--a"**, will first decrement it by one and then print its value.

The output of these arithmetic operators is shown in figure 2.

```
Following are the types of operators in C++
The value of a + b is 9
The value of a - b is -1
The value of a * b is 20
The value of a / b is 0
The value of a % b is 4
The value of a ++ is 4
The value of a -- is 5
The value of ++a is 5
The value of --a is 4
```

Figure 2: Arithmetic Operators Output

Assignment Operators

Assignment operators are used for assigning values to variables. For example: **int a = 10, b = 5;**

Comparison Operators

Comparison operators are used for comparing two values. Examples of comparison operators are shown in figure 3.

```
// Comparison operators
cout<<"Following are the comparison operators in C++"<<endl;
cout<<"The value of a == b is "<<(a==b)<<endl;
cout<<"The value of a != b is "<<(a!=b)<<endl;
cout<<"The value of a >= b is "<<(a>=b)<<endl;
cout<<"The value of a <= b is "<<(a<=b)<<endl;
cout<<"The value of a > b is "<<(a>b)<<endl;
cout<<"The value of a < b is "<<(a<b)<<endl;
```

Figure 3: Comparison Operators

1. The function "**(a==b)"**, will compare a and b values and check if they are equal. The output will be one if equal, and 0 if not.
2. The function "**(a!=b)"**, will compare a and b values and check if "a" is not equal to "b". The output will be one if not equal and 0 if equal.
3. The function "**(a>=b)"**, will compare a and b values and check if "a" is greater than or equal to "b". The output will be one if greater or equal, and 0 if not.
4. The function "**(a<=b)"**, will compare a and b values and check if "b" is greater than or equal to "a". The output will be one if greater or equal, and 0 if not.
5. The function "**(a>b)"**, will compare a and b values and check if "a" is greater than "b". The output will be one if greater and 0 if not.
6. The function "**(a<b)"**, will compare a and b values and check if "b" is greater than "a". The output will be one if greater and 0 if not.

The output of these comparison operators is shown in figure 4.

```
Following are the comparison operators in C++
The value of a == b is 0
The value of a != b is 1
The value of a >= b is 0
The value of a <= b is 1
The value of a > b is 0
The value of a < b is 1
```

Figure 4: Comparison Operators Output

Logical Operators

Logical operators are used for comparing two expressions. For example ((a==b) && (a>b)). More examples of logical operators are shown in figure 5.

```
// Logical operators
cout<<"Following are the logical operators in C++"<<endl;
cout<<"The value of this logical and operator ((a==b) && (a<b)) is:"<<((a==b) && (a<b))<<endl;
cout<<"The value of this logical or operator ((a==b) || (a<b)) is:"<<((a==b) || (a<b))<<endl;
cout<<"The value of this logical not operator (!(a==b)) is:"<<(!(a==b))<<endl;
```

Figure 5: Logical Operators

1. The function "**((a==b)&& (a<b))"** will first compare a and b values and check if they are equal or not; if they are equal, the next expression will check whether "a" is smaller than "b". The output will be one if both expressions are true and 0 if not.
2. The function "**((a==b) || (a<b))"**, will first compare a and b values and check if they are equal or not, even if they are not equal it will still check the next expression ie whether "a" is smaller than "b" or not. The output will be one if any one of the expressions is true and 0 if both are false.
3. The function "(!**(a==b))"**, will first compare a and b values and check if they are equal or not. The output will be inversed ie if "a" and "b" are equal; the output will be 0 and 1 otherwise.

The output of these logical operators is shown in figure 6.

```
Following are the logical operators in C++
The value of this logical and operator ((a==b) && (a<b)) is:0
The value of this logical or operator ((a==b) || (a<b)) is:1
The value of this logical not operator (!(a==b)) is:1
```

Figure 6: Logical Operators Output

That's it for this tutorial. In this lecture, we have covered some important operators in C++ language, but there are still some operators left, which we will cover in upcoming tutorials.

Code as described/written in the video

```
// There are two types of header files:
// 1. System header files: It comes with the compiler
include<iostream>
// 2. User defined header files: It is written by the programmer
// include "this.h" //--> This will produce an error if this.h is no present in the current directory

using namespace std;

int main(){
    int a=4, b=5;
    cout<<"Operators in C++:"<<endl;
    cout<<"Following are the types of operators in C++"<<endl;
    // Arithmetic operators
    cout<<"The value of a + b is "<<a+b<<endl;
    cout<<"The value of a - b is "<<a-b<<endl;
    cout<<"The value of a * b is "<<a*b<<endl;
    cout<<"The value of a / b is "<<a/b<<endl;
    cout<<"The value of a % b is "<<a%b<<endl;
    cout<<"The value of a++ is "<<a++<<endl;
    cout<<"The value of a-- is "<<a--<<endl;
    cout<<"The value of ++a is "<<++a<<endl;
```

```
    cout<<"The value of --a is "<<--a<<endl;
    cout<<endl;

    // Assignment Operators --> used to assign values to variables
    // int a =3, b=9;
    // char d='d';

    // Comparison operators
    cout<<"Following are the comparison operators in C++"<<endl;
    cout<<"The value of a == b is "<<(a==b)<<endl;
    cout<<"The value of a != b is "<<(a!=b)<<endl;
    cout<<"The value of a >= b is "<<(a>=b)<<endl;
    cout<<"The value of a <= b is "<<(a<=b)<<endl;
    cout<<"The value of a > b is "<<(a>b)<<endl;
    cout<<"The value of a < b is "<<(a<b)<<endl;

    // Logical operators
    cout<<"Following are the logical operators in C++"<<endl;
    cout<<"The value of this logical and operator ((a==b) && (a<b)) is:"<<((a==b) && (a<b))<<endl;
    cout<<"The value of this logical or operator ((a==b) || (a<b)) is:"<<((a==b) || (a<b))<<endl;
    cout<<"The value of this logical not operator (!(a==b)) is:"<<(!(a==b))<<endl;



    return 0;
}
```

Copy

← PreviousNext →

C++ Reference Variables & Typecasting | 7

In this C++ tutorial, we will discuss the reference variables and typecasting. In our last lesson, we discussed the header files and operators in C++. These are the topics which we are going to cover in this tutorial:

- **Built-in Data Types**
- **Float, Double and Long Double Literals**
- **Reference Variables**
- **Typecasting**

Built-in Data Types

As discussed in our previous lectures, built-in data types are pre-defined by the language and can be used directly. An example program for built-in data types is shown in figure 1.

```
5  int c = 45;
6
7  int main(){
8      int a, b, c;
9      cout<<"Enter the value of a:"<<endl;
10     cin>>a;
11     cout<<"Enter the value of b:"<<endl;
12     cin>>b;
13     c = a + b;
14     cout<<"The sum is "<<c<<endl;
15     cout<<"The global c is "<<::c;
16
```

Figure 1: Built-in Data Types

The code of built-in data types can be seen in figure 1 where we have declared three variables "**a, b and c"** inside the main function and one variable "**c"** outside the main function which is a global variable. To access the value of the global variable "**c**," we use **scope resolution operator** "**::"** with the "**c"** variable. The output of the following program is shown in figure 2.

```
Enter the value of a:
5
Enter the value of b:
9
The sum is 14
The global c is 45
```

Figure 2: Built-in Data Types Output

As we have entered the value of the variable "**a"** as five and "**b"** as 6, it gives us the sum 14, but for the global variable, it has given us the value 45.

Float, Double and Long Double Literals

The main reason to discuss these literals was to tell you an important concept about them. The float, double and long double literals program is shown in figure 3.

```
float d=34.4F;
long double e = 34.4L;
cout<<"The size of 34.4 is "<<sizeof(34.4)<<endl;
cout<<"The size of 34.4f is "<<sizeof(34.4f)<<endl;
cout<<"The size of 34.4F is "<<sizeof(34.4F)<<endl;
cout<<"The size of 34.4l is "<<sizeof(34.4l)<<endl;
cout<<"The size of 34.4L is "<<sizeof(34.4L)<<endl;
```

Figure 3: Float, Double & Long Double Literals

So the crucial concept which I am talking about is that in C++ language, double is the default type given to a decimal literal (34.4 is double by default and not float), so to use it as float, you have to specify it like "**34.4F**," as shown in figure 3. To display the size of float, double, and long double literals, we have used a "**sizeof"** operator. The output of this program is shown in figure 4.

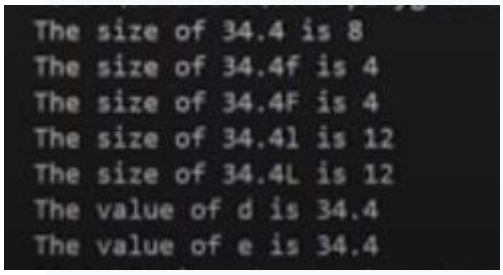


Figure 4: Float, Double, Long Double Literal Output

Reference Variable

Reference variables can be defined as another name for an already existing variable. These are also called an alias. For example, let us say we have a variable with the name of "**sum",** but we also want to use the same variable with the name of "**add"**, to do that we will make a reference variable with the name of "**add"**. The example code for the reference variable is shown in figure 5.

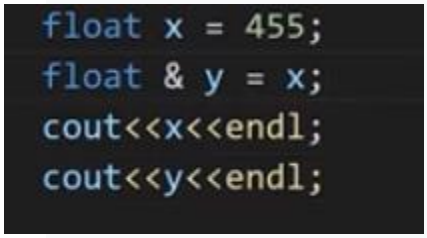


Figure 5: Reference Variable Code

As shown in figure 5, we initialized a variable "**x"** with the value "**455"**. Then we assigned the value of "**x"** to a reference variable "**y"**. The ampersand "**&"** symbol is used with the "**y"** variable to make it reference variable. Now the variable "**y"** will start referring to the value of the variable "**x"**. The output for variable "**x"** and "**y"** is shown in figure 6.

Figure 6: Reference Variable Code Output

Typecasting

Typecasting can be defined as converting one data type into another. Example code for type casting is shown in figure 7.

```
// **************Typecasting*****************
int a = 45;
float b = 45.46;
cout<<"The value of a is "<<(float)a<<endl;
cout<<"The value of a is "<<float(a)<<endl;

cout<<"The value of b is "<<(int)b<<endl;
cout<<"The value of b is "<<int(b)<<endl;
int c = int(b);

cout<<"The expression is "<<a + b<<endl;
cout<<"The expression is "<<a + int(b)<<endl;
cout<<"The expression is "<<a + (int)b<<endl;
```

Figure 7: Typecasting Example Code

As shown in figure 7, we have initialized two variables, integer "**a"** and float "**b"**. After that, we converted an integer variable "**a"** into a float variable and float variable "**b"** into an integer variable. In C++, there are two ways to typecast a variable, either using "**(float)a"** or using "**float(a)"**. The output for the above program is shown in figure 8.

```
PS D:\Business\code playground\C++ course>
The value of a is 45
The value of a is 45
The value of b is 45
The value of b is 45
The expression is 90.46
The expression is 90
The expression is 90
```

Figure 8: Typecasting Program Output

Code as described/written in the video

```
include<iostream>


using namespace std;


int c = 45;


int main(){


    // *************Build in Data types****************
    // int a, b, c;
    // cout<<"Enter the value of a:"<<endl;
    // cin>>a;
    // cout<<"Enter the value of b:"<<endl;
    // cin>>b;
    // c = a + b;
    // cout<<"The sum is "<<c<<endl;
```

```
    // cout<<"The global c is "<<::c;


    // ************* Float, double and long double Literals***************
    // float d=34.4F;
    // long double e = 34.4L;
    // cout<<"The size of 34.4 is "<<sizeof(34.4)<<endl;
    // cout<<"The size of 34.4f is "<<sizeof(34.4f)<<endl;
    // cout<<"The size of 34.4F is "<<sizeof(34.4F)<<endl;
    // cout<<"The size of 34.4l is "<<sizeof(34.4l)<<endl;
    // cout<<"The size of 34.4L is "<<sizeof(34.4L)<<endl;
    // cout<<"The value of d is "<<d<<endl<<"The value of e is "<<e;


    // *************Reference Variables***************
    // Rohan Das----> Monty -----> Rohu ------> Dangerous Coder
    // float x = 455;
    // float & y = x;
    // cout<<x<<endl;
    // cout<<y<<endl;


    // *************Typecasting***************
    int a = 45;
    float b = 45.46;
    cout<<"The value of a is "<<(float)a<<endl;
    cout<<"The value of a is "<<float(a)<<endl;


    cout<<"The value of b is "<<(int)b<<endl;
    cout<<"The value of b is "<<int(b)<<endl;
    int c = int(b);


    cout<<"The expression is "<<a + b<<endl;
    cout<<"The expression is "<<a + int(b)<<endl;
    cout<<"The expression is "<<a + (int)b<<endl;



    return 0;
}
```

Copy

Constants, Manipulators & Operator Precedence | 8

In this series of our C++ tutorials, we will visualize the constants, manipulator, and operator precedence in C++ language in this lecture. In our last lesson, we discussed the reference variable and typecasting in C++. If you haven't read the previous tutorial, click (**ADD THE LINK OF the PREVIOUS LECTURE**).

In this C++ tutorial, the topics which we are going to cover today are given below:

- **Constants in C++**
- **Manipulator in C++**
- **Operator Precedence in C++**

Constants in C++

Constants are unchangeable; when a constant variable is initialized in a program, its value cannot be changed afterwards. An example program for constants is shown in figure 1.

```
9     // Constants in C++
10    const float a = 3.11;
11    cout<<"The value of a was: "<<a<<endl;
12    a = 45.6;
13    cout<<"The value of a is: "<<a<<endl;
14    return 0;
15 }
```

Figure 1: Constants in C++

As shown in figure 2, a constant float variable "a" is initialized with a value "3.11" but when we tried to update the value of "a" with a value of "45.6" the compiler throw us an error that the constant variable is being reassigned a value. An error message can be seen in figure 2.

```
tut8.cpp: In function 'int main()':
tut8.cpp:12:9: error: assignment of read-only variable 'a'
     a = 45.6;
```

Figure 2: Constant Program Error

Manipulator

In C++ programming, language manipulators are used in the formatting of output. The two most commonly used manipulators are: "**endl"** and "**setw"**.

- "**endl"** is used for the next line.
- "**setw"** is used to specify the width of the output.

An example program to show the working of a manipulator is shown in figure 3.

```
int a =3, b=78, c=1233;
cout<<"The value of a without setw is: "<<a<<endl;
cout<<"The value of b without setw is: "<<b<<endl;
cout<<"The value of c without setw is: "<<c<<endl;

cout<<"The value of a is: "<<setw(4)<<a<<endl;
cout<<"The value of b is: "<<setw(4)<<b<<endl;
cout<<"The value of c is: "<<setw(4)<<c<<endl;
return 0;
```

Figure 3: Manipulators in C++

As shown in figure 3, we have initialized three integer variables "**a, b, c"**. First, we printed all the three variables and used "**endl"** to print each variable in a new line. After that, we again printed the three variables and used "**setw(4)**," which will set there width to "**4"**. The output for the following program is shown in figure 4.

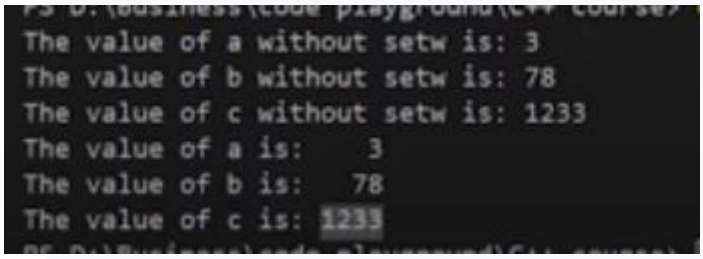


Figure 4: Manipulators Program Output

Operator Precedence & Operator Associativity

Operator precedence helps us to solve an expression. For example, in an expression "**int c = a*b+c"** the multiplication operator's precedence is higher than the precedence of addition operator, so the multiplication between "**a & b"** first and then addition will be performed.

Operator associativity helps us to solve an expression; when two or more operators have the same precedence, the operator associativity helps us to decide that we should solve the expression from "**left-to-right"** or from "**right-to-left"**.

Operator precedence and operator associativity can be seen from here. An example program for operator precedence and operator associativity is shown in figure 5.

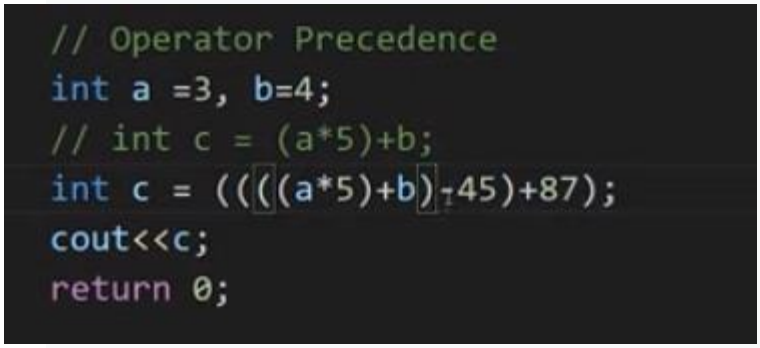


Figure 5: Operator Precedence & Associativity Example program

As shown in figure 5, we initialized two integer variables and then wrote an expression "**int c = a*5+b;"** on which we have already discussed. Then we have written another expression "**int c = ((((a*5)+b)-45)+87);".** The precedence of multiply is higher than addition so the multiplication will be done first, but the precedence of addition and subtraction is same, so here we will check the associativity which is "**left-to-right"** so the addition is performed first and then subtraction is performed.

Code as described/written in the video

```
include<iostream>

include<iomanip>


using namespace std;


int main(){

    // int a = 34;

    // cout<<"The value of a was: "<<a;

    // a = 45;

    // cout<<"The value of a is: "<<a;

    // Constants in C++

    // const int a = 3;
```

```
    // cout<<"The value of a was: "<<a<<endl;
    // a = 45; // You will get an error because a is a constant
    // cout<<"The value of a is: "<<a<<endl;

    // Manipulators in C++
    // int a =3, b=78, c=1233;
    // cout<<"The value of a without setw is: "<<a<<endl;
    // cout<<"The value of b without setw is: "<<b<<endl;
    // cout<<"The value of c without setw is: "<<c<<endl;

    // cout<<"The value of a is: "<<setw(4)<<a<<endl;
    // cout<<"The value of b is: "<<setw(4)<<b<<endl;
    // cout<<"The value of c is: "<<setw(4)<<c<<endl;



    // Operator Precedence
    int a =3, b=4;
    // int c = (a*5)+b;
    int c = ((((a*5)+b)-45)+87);
    cout<<c;
    return 0;
}
```

			/
		increment and decrement pting union member access union member access through pointer eral(C99)	
		ent and decrement[more 1] id minus and bitwise NOT	

		ereference) quirement(C11)	
		division, and remainder	
		subtraction	
		ift and right shift	
		operators < and ≤ respectively operators > and ≥ respectively	
		= and ≠ respectively	
		(exclusive or)	
		nclusive or)	
		itional[note 3]	
		nment y sum and difference y product, quotient, and remainder	

		y bitwise left shift and right shift y bitwise AND, XOR, and OR	

C++ Control Structures, If Else and Switch-Case Statement | 9

In this series of our C++ tutorials, we will visualize the control structure, if-else, and switch statements in the C++ language in this lecture. In our last lesson, we discussed the constant, manipulators and operator precedence in C++.

In this C++ tutorial, the topics which we are going to cover today are given below:

- **Control Structures in C++**
- **IF Else in C++**
- **Switch Statement in C++**

Control Structures in C++

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

1. **Sequence Structure**

Sequence structure refers to the sequence in which program execute instructions one after another. An example diagram for the sequence structure is shown in figure 1.

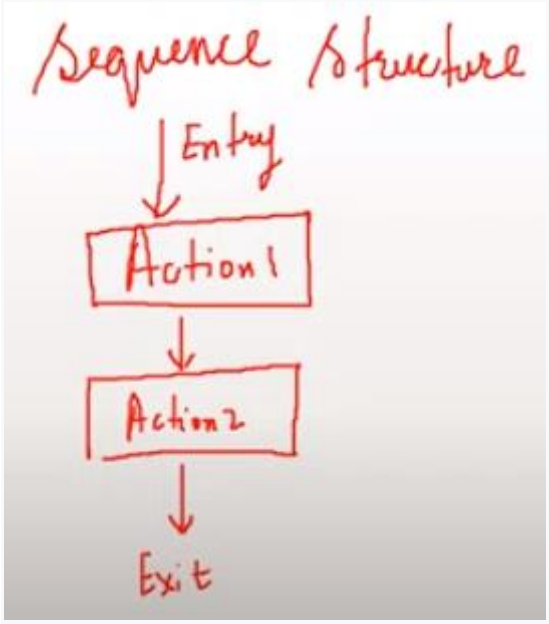


Figure 1: Sequence Structure

2. **Selection Structure**

Selection structure refers to the execution of instruction according to the selected condition, which can be either true or false. There are two ways to implement selection structures, by "**if-else statements**" or by "**switch case statements**". An example diagram for selection structure is shown in figure 2.

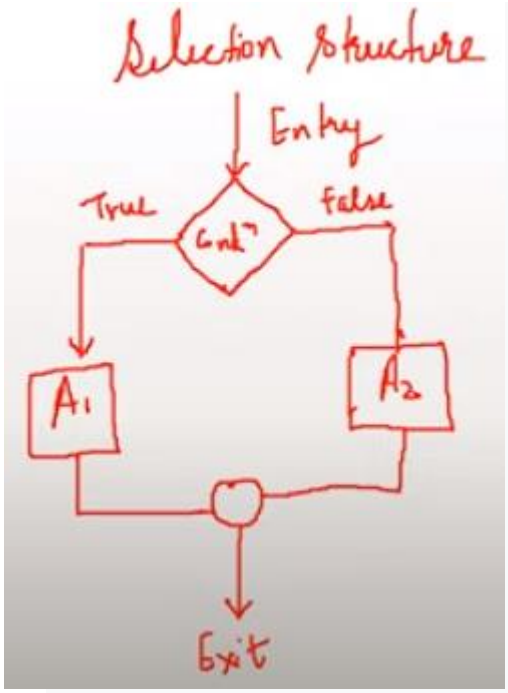


Figure 2: Selection Structure

3. **Loop Structure**

Loop structure refers to the execution of an instruction in a loop until the condition gets false. An example diagram for loop structure is shown in figure 3.

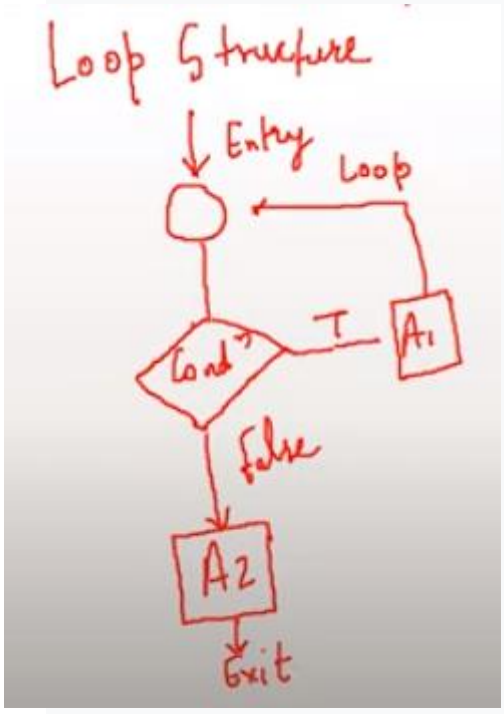


Figure 3: Loop Structure

If Else Statements in C++

As we have discussed the concepts of the different control structure, If else statements are used to implement a selection structure. An example program for if-else is shown in figure 4.

```
7    int age;
8    cout<< "Tell me your age"<<endl;
9    cin>>age;
10   if(age<18){
11       cout<<"You can not come to my party"<<endl;
12   }
13   else if(age==18){
14       cout<<"You are a kid and you will get a kid pass to the party"<<endl;
15   }
16   else{
17       cout<<"You can come to the party"<<endl;
18   }
```

Figure 4: If-Else program in C++

As shown in figure 4, we declared a variable "**age**" and used "**cin**" function to gets its value from the user at run time. At line 10 we have used "**if**" statement and give a condition "**(age<18)**" which means that if the age entered by the user is smaller than "**18**" the output will be "**you cannot come to my party**" but if the age is not smaller than "**18**" the compiler will move to the next condition.

At line 13 we have used "**else if**" statement and given another condition "**age==18**" which means that if the age entered by the user is equal to "**18**" the output will be "**you are a kid and you will get a kid pass to the party**" but if the age is not equal to the "**18**" the compiler will move to the next condition.

At line 16 we have used "**else**" condition which means that if none of the above condition is "true" the output will be "**you can come to the party**".

The output for the following program is shown in figure 5.

```
PS D:\Business\code playgrounc
Tell me your age
81
You can come to the party
```

Figure 5: If-Else Program Output

As can be seen in figure 5, that when we entered the age "**81**" which was greater than 18, so it gives us the output "**you can come to the party**". The main thing to note here is that we can use as many "**else if**" statements as we want.

Switch Case Statements in C++

In switch-case statements, the value of the variable is tested with all the cases. An example program for the switch case statement is shown in figure 6.

```
26      switch (age)
27      {
28      case 18:
29          cout<<"You are 18"<<endl;
30          break;
31      case 22:
32          cout<<"You are 22"<<endl;
33          break;
34      case 2:
35          cout<<"You are 2"<<endl;
36          break;
37
38      default:
39      cout<<"No special cases"<<endl;
40          break;
41      }
42
43      cout<<"Done with switch case";
```

Figure 6: Switch Case Statement Program

As shown in figure 4, we passed a variable "**age**" to the switch statement. The switch statement will compare the value of variable "**age**" with all cases. For example, if the entered value for variable "**age**" is "**18**", the case with value "**18**" will be executed and prints "**you are 18**". The keyword "**break**" will let the compiler skips all other cases and goes out of the switch case statement. An output of the following program is shown in figure 6.

```
Tell me your age
18
You are 18
Done with switch case
```

Figure 7: Switch Case Statement Program Output

As shown in figure 7, we entered the value "**18**" for the variable "**age**", and it gives us an output "**you are 18**" and "**Done with switch case**". The main thing to note here is that after running the "**case 18**" is skips all the other cases due to the "**break**" statement and printed "**Done with switch case**" which was outside of the switch case statement.

Code as described/written in the video

```
include<iostream>


using namespace std;


int main(){
    // cout<<"This is tutorial 9";
    int age;
    cout<< "Tell me your age"<<endl;
    cin>>age;


    // 1. Selection control structure: If else-if else ladder
    // if((age<18) && (age>0)){
```

```
    //     cout<<"You can not come to my party"<<endl;
    // }
    // else if(age==18){
    //     cout<<"You are a kid and you will get a kid pass to the party"<<endl;
    // }
    // else if(age<1){
    //     cout<<"You are not yet born"<<endl;
    // }
    // else{
    //     cout<<"You can come to the party"<<endl;
    // }

    // 2. Selection control structure: Switch Case statements
    switch (age)
    {
    case 18:
        cout<<"You are 18"<<endl;
        break;
    case 22:
        cout<<"You are 22"<<endl;
        break;
    case 2:
        cout<<"You are 2"<<endl;
        break;

    default:
    cout<<"No special cases"<<endl;
        break;
    }

    cout<<"Done with switch case";
    return 0;
}
```

Copy

For, While and do-while loops in C++ | 10

For, While and Do-While Loops in C++

In this series of our C++ tutorials, we will visualize for loop, while loop, and do-while loop in C++ language in this lecture. In our last lesson, we discussed the control structures, If-else statements, and switch statements in C++.

Loops in C++

Loops are block statements, which keeps on repeatedly executing until a specified condition is met. There are three types of loops in C++

- **For loop in C++**
- **While loop in C++**
- **Do While in C++**

For Loop in C++

For loop help us to run some specific code repeatedly until the specified condition is met. An example program **for the loop** is shown in figure 1.

```
for (int i = 0; i < 4; i++)
{
    /* code */
    cout<<i<<endl;
}
```

Figure 1: For Loop Program

As shown in figure 1, we created **for loop,** and inside its condition, there are three statements separated by a semicolon. The 1st statement is called "**initialization**", the 2nd statement is called "**condition**", and the 3rd statement is called "**updation**". After that, there is a loop body in which code is written, which needs to be repeated. Here is how our for loop will be executed:

- Initialize integer variable "**i**" with value "**0**"
- Check the condition if the value of the variable "**i**" is smaller than "**4**"
- If the condition is true go into loop body and execute the code
- Update the value of "**i**" by one
- Keep repeating this step until the condition gets false

The output for the following program is shown in figure 2.

```
0
1
2
3
```

Figure 2: For Loop Program Output

While Loop in C++

While loop helps us to run some specific code repeatedly until the specified condition is met. An example program of **while loop** is shown in figure 3.

```
int i=1;
while(i<=40){
    cout<<i<<endl;
    i++;
}
```

Figure 3: While Loop Program

As shown in figure 3, we created **a while loop,** and inside its condition, there is one statement. The statement is called "**condition**". Here is how our while loop will be executed:

- Initialize integer variable "**i**" with value "**1**"
- Check the condition if the value of the variable "**i**" is smaller or equal to "40."
- If the condition is true to go into loop body and execute the code
- Update the value of "**i**" by one
- Keep repeating this step until the condition gets false.

Do-While Loop in C++

The do-while loop helps us to run some specific code repeatedly until a specified condition is met. An example program of the **do-while loop** is shown in figure 1.

```
int i=1;
do{
    cout<<i<<endl;
    i++;
}while(i<=40);
```

Figure 4: Do-While Loop Program

As shown in figure 4, we created a **do-while loop,** and the syntax of the do-while loop is like write body with "**do**" keyword and at the end of body write "**while**" keyword with the condition. Here is how our do-while loop will be executed:

- Initialize integer variable "**i**" with value "**1**"
- Go into loop body and execute the code
- Check the condition if the value of the variable "**i**" is smaller or equal to "**40**"
- If the condition is true - go into loop body and execute the code
- Keep repeating this step until the condition gets false

Code as described/written in the video

```
include <iostream>


using namespace std;
int main()
{
    /*Loops in C++:
    There are three types of loops in C++:
        1. For loop
        2. While Loop
        3. do-While Loop
    */
```

```
/*For loop in C++*/
// int i=1;
// cout<<i;
// i++;

// Syntax for for loop
// for(initialization; condition; updation)
// {
//     loop body(C++ code);
// }

// for (int i = 1; i <= 40; i++)
// {
//     /* code */
//     cout<<i<<endl;
// }

// Example of infinite for loop
// for (int i = 1; 34 <= 40; i++)
// {
//     /* code */
//     cout<<i<<endl;
// }

/*While loop in C++*/
// Syntax:
// while(condition)
// {
//     C++ statements;
// }

//  Printing 1 to 40 using while loop
// int i=1;
// while(i<=40){
//     cout<<i<<endl;
//     i++;
// }

// Example of infinite while loop
// int i = 1;
// while (true)
// {
//     cout << i << endl;
//     i++;
// }

/* do While loop in C++*/
    // Syntax:
    // do
```

```
        // {
        //     C++ statements;
        // }while(condition);


    //  Printing 1 to 40 using while loop
    // int i=1;
    // do{
    //     cout<<i<<endl;
    //     i++;
    // }while(false);

    return 0;
}
```

Copy

Break and Continue Statements in C++ | 11

In this series of our C++ tutorials, we will visualize Break and continue statements in C++ language in this lecture. In our last lesson, we discussed for loop, while loop and do-while loop structures in C++.

In this C++ tutorial, the topics which we are going to cover today are given below:

- **Break Statements in C++**
- **Continue Statements in C++**

Break Statements

We had already discussed a little bit about break statements in switch statements. Today we will see the working of break statements in loops. Break statements in loops are used to terminate the loop. An example program for Break's statement is shown in figure 1.

```
4   int main(){
5       for (int i = 0; i < 40; i++)
6       {
7           /* code */
8           cout<<i<<endl;
9           if(i==2){
10              break;
11          }
12      }
13
```

Figure 1: Break Statement Program

As shown in figure 1, this is how the break statement program will be executed:

- Initialize integer variable "**i**" with value "**0**"
- Check the condition if the value of the variable "**i**" is smaller than "**40**"
- If the condition is true go into the loop body
- Execute "**cout**" function
- Check the condition if the value of the variable "**i**" is equal to "**2**", if it is equal terminate the loop and get out of loop body
- Update the value of "**i**" by one
- Keep repeating these steps until the loop condition gets false, or the "if" condition inside the loop body gets true.

The output of the following program is shown in figure 2.

```
PS D:\Business\cod
f ($?) { .\tut11 }
0
1
2
```

Figure 2: Break Statement Program output

Continue Statements in C++

Continue statements are somewhat similar to break statements. The main difference is that the break statement entirely terminates the loop, but the continue statement only terminates the current iteration. An example program for continue statements is shown in figure 3.

```
for (int i = 0; i < 40; i++)
{
    /* code */
    if(i==2){
        continue;
    }
    cout<<i<<endl;
}
```

Figure 3: Continue Statement Program

As shown in figure 3, this is how the continue statement program will be executed:

- Initialize integer variable "**i**" with value "**0**"
- Check the condition if the value of the variable "**i**" is smaller than "**40**"
- If the condition is true go into the loop body
- Check the condition if the value of the variable "**i**" is equal to "**2**", if it is equal terminate the loop for the current iteration and go to the next iteration
- Execute "**cout**" function
- Update the value of "**i**" by one
- Keep repeating these steps until the loop condition gets false.

Code as described/written in the video

```
include<iostream>
using namespace std;


int main(){
    // for (int i = 0; i < 40; i++)
    // {
    //     /* code */
    //     if(i==2){
    //         break;
    //     }
    //     cout<<i<<endl;
    // }
    for (int i = 0; i < 40; i++)
    {
        /* code */
```

```
        if(i==2){
            continue;
        }
        cout<<i<<endl;
    }


    return 0;
}
```

Copy

Pointers in C++ | 12

In this series of our C++ tutorials, we will visualize pointers in the C++ language in this lecture. In our last lesson, we discussed break statements and continue statements in C++.

Pointers in C++

A pointer is a data type which holds the address of other data type. The "**&**" operator is called "**address off**" operator, and the "*****" operator is called "**value at**" dereference operator. An example program for pointers is shown in figure 1.

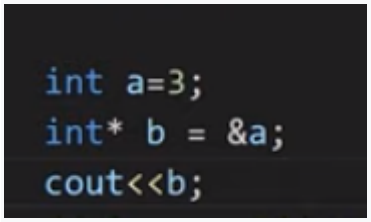


Figure 1: Pointer Program

As shown in figure 1, at 1st line an integer variable "**a**" is initialized with the value "**3**". At the 2nd line, the address of integer variable "**a**" is assigned to the integer pointer variable "**b**". At the 3rd line, the address of the integer pointer variable "**b**" is printed. The output of the following program is shown in figure 2.

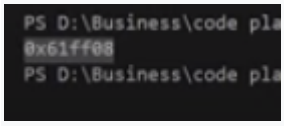


Figure 2: Pointer Program Output

As shown in figure 2, the address of the integer pointer variable "**b**" is printed. The main thing to note here is that the address printed by the variable "**b**" is the address of integer variable "**a**" because we had assigned the address of variable "**a**" to the integer pointer variable "**b**". To clarify, we will print both variable "a" and variable "b" addresses, which are shown in figure 3.

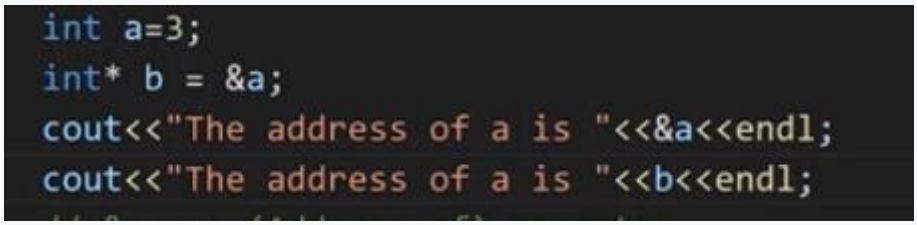


Figure 3: Pointer Program Example 2

As shown in figure 3, now we printed both variable "**a**" and variable "**b**" addresses. The output for the following program is shown in figure 4.

```
PS D:\Business\code playground
The address of a is 0x61ff08
The address of a is 0x61ff08
```

Figure 4: Pointer Program Example 2 Output

As shown in figure 4, both variables "**a**" and "**b**" have the same addresses, but in actual, this is the address of the variable "**a**", the variable "**b**" is just pointing to the address of the variable "**a**".

To see the value of variable "**a**" using a pointer variable, we can use the "*****" dereference operator. An example of the dereference operator program is shown in figure 5.

```
// * ---> (value at) Dereference operator
cout<<"The value at address b is "<<*b<<endl;
```

Figure 5: Dereference Operator example

As shown in figure 5, the value at address "**b**" is printed. The main thing to note here is that the value printed by the pointer variable "**b**" will be the value of variable "**a**" because the pointer variable "**b**" is pointing to the address of the variable "**a**". The output for the following program is shown in figure 6.

```
// * ---> (value at) Dereference operator
cout<<"The value at address b is "<<*b<<endl;
```

Figure 6: Dereference Operator Example

Pointer to Pointer

Pointer to Pointer is a simple concept, in which we store the address of one Pointer to another pointer. An example program for Pointer to Pointer is shown in figure 7.

```
// Pointer to pointer
int** c = &b;
cout<<"The address of b is "<<&b<<endl;
cout<<"The address of b is "<<c<<endl;
cout<<"The value at address c is "<<*c<<endl;
cout<<"The value at address value_at(value_at(c)) is "<<**c<<endl;
```

Figure 7: Pointer to Pointer Example Program

As shown in figure 7, at the 1st line, the address of the pointer variable "**b**" is assigned to the pointer variable "**c**". At 2nd line, the address of the pointer variable "**b**" is printed. At the 3rd line, the address of the pointer variable "**c**" is printed. At line 4th, the value at the pointer variable "**c**" is printed. At line 5th, the pointer variable "**c**" will be dereferenced two times, and it will print the value at pointer variable "**b**". The output of the following program is shown in figure 2. The output for the following program is shown in figure 8.

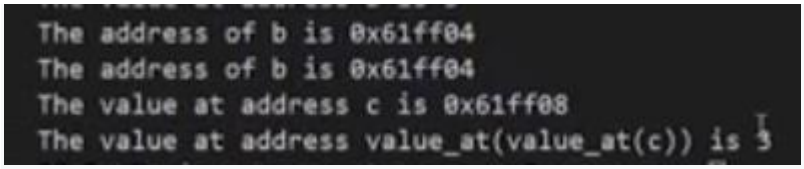


Figure 8: Pointer to Pointer Example Program Output

Code as described/written in the video

```
include<iostream>
using namespace std;

int main(){
    // What is a pointer? ----> Data type which holds the address of other data types
    int a=3;
    int* b;
    b = &a;

    // & ---> (Address of) Operator
    cout<<"The address of a is "<<&a<<endl;
    cout<<"The address of a is "<<b<<endl;

    // * ---> (value at) Dereference operator
    cout<<"The value at address b is "<<*b<<endl;

    // Pointer to pointer
    int** c = &b;
    cout<<"The address of b is "<<&b<<endl;
    cout<<"The address of b is "<<c<<endl;
    cout<<"The value at address c is "<<*c<<endl;
    cout<<"The value at address value_at(value_at(c)) is "<<**c<<endl;

    return 0;}
```

Arrays & Pointers Arithmetic in C++ | 13

In this tutorial, we will discuss arrays and pointer arithmetic in C++

What are Arrays in C++

- An array is a collection of items which are of the similar type stored in contiguous memory locations.
- Sometimes, a simple variable is not enough to hold all the data.
- For example, let's say we want to store the marks of 2500 students; initializing 2500 different variable for this task is not feasible.
- To solve this problem, we can define an array with size 2500 that can hold the marks of all students.
- For example **int marks[2500];**

An example program for an array is shown in code snippet below.

```
int marks[] = {23, 45, 56, 89};
cout<<marks[0]<<endl;
```

```
cout<<marks[1]<<endl;
cout<<marks[2]<<endl;
cout<<marks[3]<<endl;
```

Copy

Code Snippet 1: Array Program 1

As shown in the code snippet, we initialized an array of size 4 in which we have stored marks of 4 students and then printed them one by one. The main point to note here is that array store data in continuous block form in the memory, and array indexes start from 0. Output for the following program is shown in figure 1.

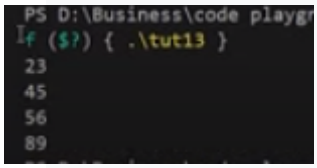


Figure 1: Array Program 1 Output

Another example program to declare an array is shown in code snippet 2.

```
int mathMarks[4];
    mathMarks[0] = 2278;
    mathMarks[1] = 738;
    mathMarks[2] = 378;
    mathMarks[3] = 578;

    cout<<"These are math marks"<<endl;
    cout<<mathMarks[0]<<endl;
    cout<<mathMarks[1]<<endl;
    cout<<mathMarks[2]<<endl;
    cout<<mathMarks[3]<<endl;
```

Copy

Code Snippet 2: Array Program 2

As shown in code snippet 2, we have declared an array of size 4 and then assigned values one by one to each index of the array. Output for the following program is shown in figure 2.

Figure 2: Array Program 2 Output

To change the value at the specific index of an array, we can simply assign the value to that index. For example: "**marks[2] = 333**" can place the value "**333**" at the index "**2**" of the array. We can use loops to print the values of an array, instead of printing them one by one. An example program to print the value of the array with "for" loop is shown in code snippet 3.

```
for (int i = 0; i < 4; i++)
    {
        cout<<"The value of marks "<<i<<" is "<<marks[i]<<endl;
    }
```

Copy
Code Snippet 3: Array program with a loop

As shown in code snippet 3, we initialized an integer variable “i" with the value 0 and set the running condition of the loop to the length of an array. In the loop body, each index number and the value at each number is being printed. Output for the following program is shown in figure 3.

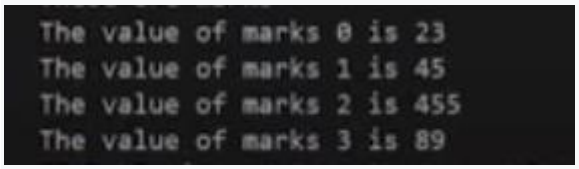


Figure 3: Array program with loop output

Pointers and Arrays

Storing the address of an array into pointer is different than storing the address of a variable into the pointer because the name of the array is an address of the first index of an array. So to use ampersand "&" with the array name for assigning the address to a pointer is wrong.

- &Marks --> Wrong
- Marks --> address of the first block

An example program for storing the starting address of an array in the pointer is shown in code snippet 4.

```
int* p = marks;

cout<<"The value of marks[0] is "<<*p<<endl;
```

Copy
Code Snippet 4: Pointer and Array Program

As shown in code snippet 7, we have assigned the address of array “marks” to the pointer variable “*p” and then printed the pointer “*p”. The main thing to note here is that the value at the pointer “*p” is the starting address of the array “marks”. The output for the following program is shown in figure 4.

Figure 4: Pointer and Array Program Output

As shown in figure 4, we have printed the value at pointer "*p", and it has shown us the value of the first index of the array "marks" because the pointer was pointing at the first index of an array and the value at that index was "23". If we want to access the 2nd index of an array through the pointer, we can simply increment the pointer with 1. For example: "***(p+1)**" will give us the value of the 2nd index of an array. An example program to print the values of an array through the pointer is shown in code snippet 5.

```
int* p = marks;

    cout<<"The value of *p is "<<*p<<endl;

    cout<<"The value of *(p+1) is "<<*(p+1)<<endl;

    cout<<"The value of *(p+2) is "<<*(p+2)<<endl;

    cout<<"The value of *(p+3) is "<<*(p+3)<<endl;
```

Copy
Code Snippet 5: Pointer and Array Program 2

As shown in code snippet 5, 1st we have printed the value at pointer “***p**”; 2nd we have printed the value at pointer “***(p+1)**”; 3rd we have printed the value at pointer “***(p+2)**”; 4th we have printed the value at pointer “***(p+3)**". This program will output the values at "0, 1, 2, 3" indices of an array "marks". The output of the following program is shown in figure 5.

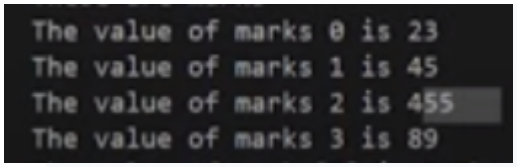


Figure 5: Pointer and Array Program 2 Output

Code as described/written in the video

```
include<iostream>
using namespace std;

int main(){
    // Array Example
    int marks[] = {23, 45, 56, 89};

    int mathMarks[4];
    mathMarks[0] = 2278;
    mathMarks[1] = 738;
    mathMarks[2] = 378;
    mathMarks[3] = 578;

    cout<<"These are math marks"<<endl;
    cout<<mathMarks[0]<<endl;
    cout<<mathMarks[1]<<endl;
    cout<<mathMarks[2]<<endl;
    cout<<mathMarks[3]<<endl;

    // You can change the value of an array
    marks[2] = 455;
    cout<<"These are marks"<<endl;
    // cout<<marks[0]<<endl;
    // cout<<marks[1]<<endl;
    // cout<<marks[2]<<endl;
```

Structures, Unions & Enums in C++ | 14

In this tutorial, we will discuss structures, unions & enums in C++

Structures in C++

The structure is a user-defined data type that is available in C++. Structures are used to combine different types of data types, just like an array is used to combine the same type of data types. An example program for creating a structure is shown in Code Snippet 1.

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

```
Copy
```

Code Snippet 1: Creating a Structure Program

As shown in Code Snippet 1, we have created a structure with the name “employee”, in which we have declared three variables of different data types (eId, favchar, salary). As we have created a structure now we can create instances of our structure employee. An example program for creating instances of structure employees is shown in Code Snippet 2.

```
int main() {
    struct employee harry;
    harry.eId = 1;
    harry.favChar = 'c';
    harry.salary = 120000000;
    cout<<"The value is "<<harry.eId<<endl;
    cout<<"The value is "<<harry.favChar<<endl;
    cout<<"The value is "<<harry.salary<<endl;
    return 0;
}
Copy
```

Code Snippet 2: Creating Structure instances

As shown in Code Snippet 2, 1st we have created a structure variable “harry” of type “employee”, 2nd we have assigned values to (eId, favchar, salary) fields of the structure employee and at the end we have printed the value of “salary”.

Another way to create structure variables without using the keyword “struct” and the name of the struct is shown in Code Snippet 3.

```
typedef struct employee
{
    /* data */
    int eId; //4
    char favChar; //1
    float salary; //4
} ep;
Copy
```

Code Snippet 3: Creating Structure Program 2

As shown in Code Snippet 3, we have used a keyword “**typedef**” before struct and after the closing bracket of structure, we have written “ep”. Now we can create structure variables without using the keyword “struct” and name of the struct. An example is shown in Code Snippet 4.

```
int main(){
ep harry;
    struct employee shubham;
    struct employee rohanDas;
    harry.eId = 1;
    harry.favChar = 'c';
    harry.salary = 120000000;
    cout<<"The value is "<<harry.eId<<endl;
    cout<<"The value is "<<harry.favChar<<endl;
    cout<<"The value is "<<harry.salary<<endl;
    return 0;
}
```

Copy

Code Snippet 4: Creating Structure instance 2

As shown in Code Snippet 4, we have created a structure instance “harry” by just writing “ep” before it.

Unions in C++

Unions are similar to structures but they provide better memory management then structures. Unions use shared memory so only 1 variable can be used at a time. An example program to create unions is shown in Code Snippet 5.

```
union money
{
    /* data */
    int rice; //4
    char car; //1
    float pounds; //4
};
```

Copy

Code Snippet 5: Creating Unions Program

As shown in Code Snippet 5, we have created a union with the name “money” in which we have declared three variables of different data types (rice, car, pound). The main thing to note here is that:

- We can only use 1 variable at a time otherwise the compiler will give us a garbage value
- The compiler chooses the data type which has maximum memory for the allocation.

An example program for creating an instance of union money is shown in Code Snippet 6.

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

Copy

Code Snippet 6: Creating a Union Instance

As shown in Code Snippet 6, 1st we have created a union variable “m1” of type “money”, 2nd we have assigned values to (rice) fields of the union money, and in the end, we have printed the value of “rice”. The main thing to note here is that once we have assigned a value to the union field “rice”, now we cannot use other fields of the union otherwise we will get garbage value. The output for the following program is shown in figure 1.

Figure 1: Creating Union Instance Output

Enums in C++

Enums are user-defined types which consist of named constants. Enums are used to make the program more readable. An example program for enums is shown in Code Snippet 8.

```
int main(){
    enum Meal{ breakfast, lunch, dinner};
    Meal m1 = lunch;
    cout<<m1;
    return 0;
}
```

Copy

Code Snippet 7: Enums Program

As shown in Code Snippet 7, 1st we have created an enum “Meal” in which we have stored three named constants (breakfast, lunch, dinner). 2nd we have assigned the value of “lunch” to the variable “m1” and at the end, we have printed “m1”. The main thing to note here is that (breakfast, lunch, dinner) are constants; the value for “breakfast” is “0”, the value for “lunch” is “1” and the value for “dinner” is “2”. The output for the following program is shown in figure 2.

Figure 2: Enums Program Output

Code as described/written in the video

```
include<iostream>
using namespace std;

typedef struct employee
{
    /* data */
    int eId; //4
    char favChar; //1
    float salary; //4
} ep;

union money
{
    /* data */
    int rice; //4
    char car; //1
    float pounds; //4
};



int main(){
    enum Meal{ breakfast, lunch, dinner};
    Meal m1 = lunch;
    cout<<(m1==2);
    // cout<<breakfast;
    // cout<<lunch;
    // cout<<dinner;
    // union money m1;
    // m1.rice = 34;
    // m1.car = 'c';
    // cout<<m1.car;

    // ep harry;
    // struct employee shubham;
    // struct employee rohanDas;
    // harry.eId = 1;
```

```
    // harry.favChar = 'c';
    // harry.salary = 120000000;
    // cout<<"The value is "<<harry.eId<<endl;
    // cout<<"The value is "<<harry.favChar<<endl;
    // cout<<"The value is "<<harry.salary<<endl;
    return 0;
}
```

Functions & Function Prototypes in C++ | 15

In this tutorial, we will discuss functions and functions prototype in C++

Functions in C++

Functions are the main part of top-down structured programming. We break the code into small pieces and make functions of that code. Functions help us to reuse the code easily. An example program for the function is shown in Code Snippet 1.

```
int sum(int a, int b){
 int c = a+b;
    return c;
}
```

Copy

Code Snippet 1: Function example

As shown in Code Snippet 1, we created an integer function with the name of sum, which takes two parameters “int a” and “int b”. In the function, body addition is performed on the values of variable “a” and variable “b” and the result is stored in variable “c”. In the end, the value of variable “c” is returned to the function. We have seen how this function works now we will see how to pass values to the function parameters. An example program for passing the values to the function is shown in Code Snippet 2.

```
int main(){
    int num1, num2;
    cout<<"Enter first number"<<endl;
    cin>>num1;
    cout<<"Enter second number"<<endl;
    cin>>num2;
    cout<<"The sum is "<<sum(num1, num2);
    return 0;
}
```

Copy

Code Snippet 2: Passing Value to Function Parameters

As shown in Code Snippet 2, we have declared two integer variables “num1” and “num2”, we will take their input at run time. In the end, we called the “sum” function and passed both variables “num1” and “num2” into sum function. “sum” function will perform the addition and returns the value at the same location from where it was called. The output of the following program is shown in figure 1.

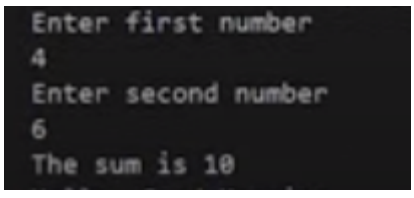


Figure 1: Function Output

Function Prototype in C++

The function prototype is the template of the function which tells the details of the function e.g(name, parameters) to the compiler. Function prototypes help us to define a function after the function call. An example of a function prototype is shown in Code Snippet 3.

```
// Function prototype
    int sum(int a, int b);
```

Copy

Code Snippet 3: Function Prototype

As shown in Code Snippet 3, we have made a function prototype of the function “sum”, this function prototype will tell the compiler that the function “sum” is declared somewhere in the program which takes two integer parameters and returns an integer value. Some examples of acceptable and not acceptable prototypes are shown below:

- int sum(int a, int b); //Acceptable
- int sum(int a, b); // Not Acceptable
- int sum(int, int); //Acceptable

Formal Parameters

The variables which are declared in the function are called a formal parameter. For example, as shown in Code Snippet 1, the variables “a” and “b” are the formal parameters.

Actual Parameters

The values which are passed to the function are called actual parameters. For example, as shown in Code Snippet 2, the variables “num1” and “num2” are the actual parameters.

The function doesn't need to have parameters or it should return some value. An example of the void function is shown in Code Snippet 4.

```
void g(){
    cout<<"\nHello, Good Morning";
}
```

Copy

Code Snippet 4: Void Function

As shown in Code Snippet 4, void as a return type means that this function will not return anything, and this function has no parameters. Whenever we will call this function it will print “Hello, Good Morning”

Code as described/written in the video

```
include<iostream>
using namespace std;


// Function prototype
// type function-name (arguments);
// int sum(int a, int b); //--> Acceptable
// int sum(int a, b); //--> Not Acceptable
int sum(int, int); //--> Acceptable
// void g(void); //--> Acceptable
void g(); //--> Acceptable


int main(){
    int num1, num2;
    cout<<"Enter first number"<<endl;
    cin>>num1;
    cout<<"Enter second number"<<endl;
```

```
    cin>>num2;
    // num1 and num2 are actual parameters
    cout<<"The sum is "<<sum(num1, num2);
    g();
    return 0;
}

int sum(int a, int b){
    // Formal Parameters a and b will be taking values from actual parameters num1 and num2.
    int c = a+b;
    return c;
}

void g(){
    cout<<"\nHello, Good Morning";
}
```

Call by Value & Call by Reference in C++ | 16

In this tutorial, we will discuss call by value and call by reference in C++

Call by Value in C++

Call by value is a method in C++ to pass the values to the function arguments. In case of call by value the copies of actual parameters are sent to the formal parameter, which means that if we change the values inside the function that will not affect the actual values. An example program for the call by value is shown in Code Snippet 1.

```
void swap(int a, int b){ //temp a b
    int temp = a;        //4   4  5
    a = b;               //4   5  5
    b = temp;            //4   5  4
}
```

Copy

Code Snippet 1: Call by Value Swap Function

As shown in Code Snippet 1, we created a swap function which is taking two parameters “int a” and “int b”. In function body values of the variable, “a” and “b” are swapped. An example program is shown in Code Snippet 2, which calls the swap function and passes values to it.

```
int main(){
    int x =4, y=5;
    cout<<"The value of x is "<<x<<" and the value of y is "<<y<<endl;
    swap(x, y);
    cout<<"The value of x is "<<x<<" and the value of y is "<<y<<endl;
    return 0;
}
```

Copy

Code Snippet 2: Passing Values to Swap Function

As shown in Code Snippet 2, we have initialized two integer variables “a” and “b” and printed their values. Then we called a “swap” function and passed values of variables “a” and “b” and again printed the values of variables “a” and “b”. The output for the following program is shown in figure 1.

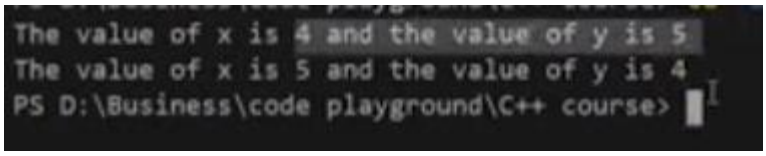


Figure 1: Call by Value Swap Function Output

As shown in figure 3, the values of “a” and “b” are the same for both times they are printed. So the main point here is that when the call by value method is used it doesn’t change the actual values because copies of actual values are sent to the function.

Call by Pointer in C++

A call by the pointer is a method in C++ to pass the values to the function arguments. In the case of call by pointer, the address of actual parameters is sent to the formal parameter, which means that if we change the values inside the function that will affect the actual values. An example program for the call by reference is shown in Code Snippet 3.

```
// Call by reference using pointers
void swapPointer(int* a, int* b){ //temp a b
    int temp = *a;          //4   4  5
    *a = *b;               //4   5  5
    *b = temp;            //4   5  4
}
```

Copy

Code Snippet 3: Call by Pointer Swap Function

As shown in Code Snippet 3, we created a swap function which is taking two pointer parameters “int* a” and “int* b”. In function body values of pointer variables, “a” and “b” are swapped. An example program is shown in Code Snippet 4, which calls the swap function and passes values to it.

```
int main(){
    int x =4, y=5;
    cout<<"The value of x is "<<x<<" and the value of y is "<<y<<endl;
    swapPointer(&x, &y);   //This will swap a and b using pointer reference
    cout<<"The value of x is "<<x<<" and the value of y is "<<y<<endl;
    return 0;
}
```

Copy

Code Snippet 4: Passing Values to Call by Pointer Swap Function

As shown in Code Snippet 4, we have initialized two integer variables “a” and “b” and printed their values. Then we called a “swap” function and passed addresses of variables “a” and “b” and again printed the values of variables “a” and “b”. The output for the following program is shown in figure 2.

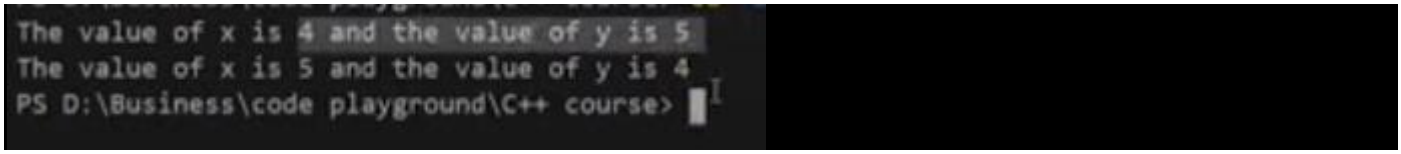


Figure 2: Call by Pointer Swap Function Output

As shown in figure 2, the values of “a” and “b” are swapped when the swap function is called. So the main point here is that when the call by pointer method is used it changes the actual values because addresses of actual values are sent to the function.

Call by Reference in C++

Call by reference is a method in C++ to pass the values to the function arguments. In the case of call by reference, the reference of actual parameters is sent to the formal parameter, which means that if we change the values inside the function that will affect the actual values. An example program for a call by reference is shown in Code Snippet 5.

```
void swapReferenceVar(int &a, int &b){    //temp a b
    int temp = a;          //4   4  5
    a = b;               //4   5  5
    b = temp;            //4   5  4
}
```

Copy

Code Snippet 5: Call by Reference Swap Function

As shown in Code Snippet 5, we created a swap function that is taking reference of “int &a” and “int &b” as parameters. In function body values of variables, “a” and “b” are swapped. An example program is shown in Code Snippet 6, which calls the swap function and passes values to it.

```
int main(){
    int x =4, y=5;
    cout<<"The value of x is "<<x<<" and the value of y is "<<y<<endl;
    swapReferenceVar(x, y); //This will swap a and b using reference variables
    cout<<"The value of x is "<<x<<" and the value of y is "<<y<<endl;
    return 0;
}
```

Copy

Code Snippet 6: Passing Values to Call by Reference Swap Function

As shown in Code Snippet 6, we have initialized two integer variables “a” and “b” and printed their values. Then we called a “swap” function and passed variables “a” and “b” and again printed the values of variables “a” and “b”. The output for the following program is shown in figure 3.

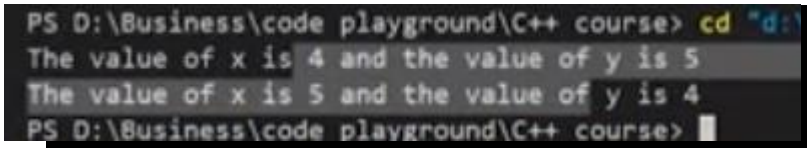


Figure 3: Call by Reference Swap Function Output

As shown in figure 3, the values of “a” and “b” are swapped when the swap function is called. So the main point here is that when the call by reference method is used it changes the actual values because references of actual values are sent to the function.

Code as described/written in the video

```
include<iostream>
using namespace std;

int sum(int a, int b){
    int c = a + b;
    return c;
}

// This will not swap a and b
void swap(int a, int b){ //temp a b
    int temp = a;        //4   4  5
    a = b;               //4   5  5
    b = temp;            //4   5  4
```

```
}


// Call by reference using pointers
void swapPointer(int* a, int* b){ //temp a b
    int temp = *a;          //4   4  5
    *a = *b;               //4   5  5
    *b = temp;            //4   5  4
}


// Call by reference using C++ reference Variables
// int &
void swapReferenceVar(int &a, int &b){ //temp a b
    int temp = a;          //4   4  5
    a = b;               //4   5  5
    b = temp;            //4   5  4
    // return a;
}


int main(){
    int x =4, y=5;
    // cout<<"The sum of 4 and 5 is "<<sum(a, b);
    cout<<"The value of x is "<<x<<" and the value of y is "<<y<<endl;
    // swap(x, y); // This will not swap a and b
    // swapPointer(&x, &y); //This will swap a and b using pointer reference
    swapReferenceVar(x, y); //This will swap a and b using reference variables
    // swapReferenceVar(x, y) = 766; //This will swap a and b using reference variables
    cout<<"The value of x is "<<x<<" and the value of y is "<<y<<endl;
    return 0;
}
```

Inline Functions, Default Arguments & Constant Arguments in C++ | 17

In this tutorial, we will discuss inline functions, default arguments, and constant arguments in C++

Inline Functions in C++

Inline functions are used to reduce the function call. When one function is being called multiply times in the program it increases the execution time, so inline function is used to reduce time and increase program efficiency. If the inline function is being used when the function is called, the inline function expands the whole function code at the point of a function call, instead of running the function. Inline functions are considered to be used when the function is small otherwise it will not perform well. Inline is not recommended when static variables are being used in the function. An example of an inline function is shown in Code Snippet 1.

```
inline int product(int a, int b){
    return a*b;
}
```

Copy

Code Snippet 1: Inline function

As shown in Code Snippet 1, 1st inline keyword is used to make the function inline. 2nd a product function is created which has two arguments and returns the product of them. Now we will call the product function multiple times in our main program which is shown in Code Snippet 2.

```
int main(){
    int a, b;
```

```
    cout<<"Enter the value of a and b"<<endl;
    cin>>a>>b;
    cout<<"The product of a and b is "<<product(a,b)<<endl;
    cout<<"The product of a and b is "<<product(a,b)<<endl;
    cout<<"The product of a and b is "<<product(a,b)<<endl;
    cout<<"The product of a and b is "<<product(a,b)<<endl;
    cout<<"The product of a and b is "<<product(a,b)<<endl;
    cout<<"The product of a and b is "<<product(a,b)<<endl;
    cout<<"The product of a and b is "<<product(a,b)<<endl;
    cout<<"The product of a and b is "<<product(a,b)<<endl;
    return 0;
}
```

Copy

Code Snippet 2: Calling Inline Product Function

As shown in Code Snippet 2, we called the product function multiple times. The main thing to note here is that the function will not run instead of it the function code will be copied at the place where the function is being called. This will increase the execution time of the program because the compiler doesn't have to copy the values and get the return value again and again from the compiler. The output of the following program is shown in figure 1.

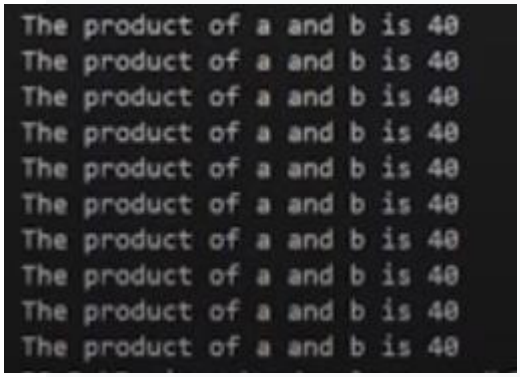


Figure 1: Inline Function Output

Default Arguments in C++

Default arguments are those values which are used by the function if we don't input our value. It is recommended to write default arguments after the other arguments. An example program for default arguments is shown in Code Snippet 3.

```
float moneyReceived(int currentMoney, float factor=1.04){
    return currentMoney * factor;
}

int main(){
    int money = 100000;
    cout<<"If you have "<<money<<" Rs in your bank account, you will recive "<<moneyReceived(money)<< "Rs after 1 year"<<endl;
    cout<<"For VIP: If you have "<<money<<" Rs in your bank account, you will recive "<<moneyReceived(money, 1.1)<< " Rs after 1 year";
    return 0;
}
```

Copy

Code Snippet 3: Default Argument Example Program

As shown in Code Snippet 3, we created a "moneyReceived" function which has two arguments "int currentMoney" and "float factor=1.04". This function returns the product of "currentMoney" and "factor". In our main function, we called "moneyReceived" function and passed one argument "money". Again we called "moneyReceived" function and passed two arguments "money" and "1.1". The main thing to note here is that when we

passed only one argument “money” to the function at that time the default value of the argument “factor” will be used. But when we passed both arguments then the default value will not be used. The output for the following program is shown in figure 2.

```
PS D:\Business\code playground\C++ course> cd "d:\Business\code playground\C++ course\" ; if ($?) { g++ tut17.cpp -o tut17 } ; if ($?) { .\tut17 }
If you have 100000 Rs in your bank account, you will recive 104000Rs after 1 yearFor VIP: If you have 100000 Rs in your bank account, you will recive 110000Rs af
ter 1 year
```

Figure 2: Default Argument Example Program Output

Constant Arguments in C++

Constant arguments are used when you don’t want your values to be changed or modified by the function. An example of constant arguments is shown in Code Snippet 4.

```
int strlen(const char *p){


}
```

Copy

Code Snippet 4: Constant Arguments Example

As shown in Code Snippet 4, we created a “strlen” function which takes a constant argument “p”. As the argument is constant so its value won’t be modified.

Code as described/written in the video

```
include<iostream>
using namespace std;


inline int product(int a, int b){
    // Not recommended to use below lines with inline functions
    // static int c=0; // This executes only once
    // c = c + 1; // Next time this function is run, the value of c will be retained
    return a*b;
}


float moneyReceived(int currentMoney, float factor=1.04){
    return currentMoney * factor;
}


// int strlen(const char *p){


// }
int main(){
    int a, b;
    // cout<<"Enter the value of a and b"<<endl;
    // cin>>a>>b;
    // cout<<"The product of a and b is "<<product(a,b)<<endl;
    int money = 100000;
    cout<<"If you have "<<money<<" Rs in your bank account, you will recive "<<moneyReceived(money)<< "Rs after 1 year"<<endl;
    cout<<"For VIP: If you have "<<money<<" Rs in your bank account, you will recive "<<moneyReceived(money, 1.1)<< " Rs after 1 year";
    return 0;
}
```

Copy

In this tutorial, we will discuss recursion and recursive functions in C++

Recursion and Recursive Function

When a function calls itself it is called recursion and the function which is calling itself is called a recursive function. The recursive function consists of a base case and recursive condition. It is very important to add a base case in recursive function otherwise recursive function will never stop executing. An example of the recursive function is shown in Code Snippet 1.

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

Copy

Code Snippet 1: Factorial Recursive Function

As shown in Code Snippet 1, we created a “factorial” function which takes one argument. In the function body, there is a base case which checks that if the value of variable “n” is smaller or equal to “1” if the condition is “true” return “1”. And there is a recursive condition that divides the bigger value to smaller values and at the end returns a factorial. These are the steps which will be performed by recursive condition:

- **4 * factorial(4-1)**
- **4 * 3 * factorial(3-1)**
- **4* 3 * 2 * factorial(2-1)**
- **4 * 3 * 2 * 1**

An example to pass the value to the recursive factorial function is shown in Code Snippet 2.

```
int main(){
    int a;
    cout<<"Enter a number"<<endl;
    cin>>a;
    cout<<"The factorial of "<<a<< " is "<<factorial(a)<<endl;
    return 0;
}
```

Copy

Code Snippet 2: Factorial Recursive Function Call

As shown in Code Snippet 2, we created an integer variable “a”, which takes input at the runtime and that value is passed to the factorial function. The output for the following program is shown in figure 1.

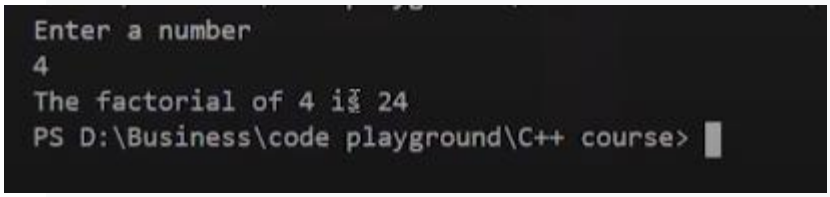


Figure 1: Factorial Recursive Function Output

As shown in figure 1, we input the value “4” and it gives us the factorial of it which is “24”. Another example of a recursive function for the Fibonacci series is shown in Code Snippet 3.

```
int fib(int n){
    if(n<2){
        return 1;
```

```
    }
    return fib(n-2) + fib(n-1);
}
```

Copy

Code Snippet 3: Fibonacci Recursive Function

As shown in Code Snippet 3, we created a “fib” function which takes one argument. In the function body, there is a base case which checks that if the value of variable “n” is smaller than “2”, if the condition is “true” return “1”. And there is a recursive condition that divides the bigger value to smaller values and at the end returns a Fibonacci number. An example to pass the value to the Fibonacci function is shown in Code Snippet 4.

```
int main(){
    int a;
    cout<<"Enter a number"<<endl;
    cin>>a;
    cout<<"The term in fibonacci sequence at position "<<a<< " is "<<fib(a)<<endl;
    return 0;
}
```

Copy

Code Snippet 4: Fibonacci Recursive Function Call

As shown in Code Snippet 4, we created an integer variable “a”, which takes input at the runtime and that value is passed to the Fibonacci function. The output for the following program is shown in figure 2.

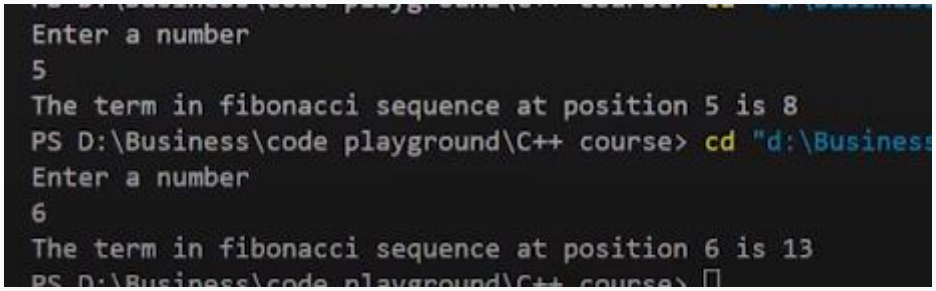


Figure 2: Fibonacci Recursive Function Output

As shown in figure 2, 1st we input the value “5” and it gives us the Fibonacci number at that place which is “8”. 2nd we input the value “6” and it gives us the Fibonacci number at that place which is “13”.

One thing to note here is that recursive functions are not always the best option. They perform well in some problems but not in every problem.

Code as described/written in the video

```
include<iostream>
using namespace std;

int fib(int n){
    if(n<2){
        return 1;
    }
    return fib(n-2) + fib(n-1);
}

// fib(5)
// fib(4) + fib(3)
// fib(2) + fib(3) + fib(2) + fib(3)
```

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

// Step by step calculation of factorial(4)
// factorial(4) = 4 * factorial(3);
// factorial(4) = 4 * 3 * factorial(2);
// factorial(4) = 4 * 3 * 2 * factorial(1);
// factorial(4) = 4 * 3 * 2 * 1;
// factorial(4) = 24;

int main(){
    // Factorial of a number:
    // 6! = 6*5*4*3*2*1 = 720
    // 0! = 1 by definition
    // 1! = 1 by definition
    // n! = n * (n-1)!
    int a;
    cout<<"Enter a number"<<endl;
    cin>>a;
    // cout<<"The factorial of "<<a<< " is "<<factorial(a)<<endl;
    cout<<"The term in fibonacci sequence at position "<<a<< " is "<<fib(a)<<endl;
    return 0;
}
```

Copy

Function Overloading with Examples in C++ | 19

In this tutorial, we will discuss function overloading in C++

Function Overloading in C++

Function overloading is a process to make more than one function with the same name but different parameters, numbers, or sequence. An example program to explain function overloading is shown in Code Snippet 1.

```
int sum(float a, int b){
    cout<<"Using function with 2 arguments"<<endl;
    return a+b;
}

int sum(int a, int b, int c){
    cout<<"Using function with 3 arguments"<<endl;
    return a+b+c;
}
```

Copy

Code Snippet 1: Sum Function Overloading Example

As shown in Code Snippet 1, we have created two “sum” functions, the 1st “sum” function takes two arguments “int a”, “int b” and return the sum of those two variables; and the 2nd sum function is taking three arguments “int a”, “int b”, “int c” and return the sum of those three variables. Function call for these “sum” function is shown in Code Snippet 2.

```
int main(){
    cout<<"The sum of 3 and 6 is "<<sum(3,6)<<endl;
    cout<<"The sum of 3, 7 and 6 is "<<sum(3, 7, 6)<<endl;
    return 0;
}
```

Copy

Code Snippet 2: Sum Function Call

As shown in Code Snippet 2, we passed two arguments in the first function call and three arguments in the second function call. The output of the following program is shown in figure 1.

```
The sum of 3 and 6 is Using function with 2 arguments
9
The sum of 3, 7 and 6 is Using function with 3 arguments
16
```

Figure 1: Sum Function Output

As shown in Code Snippet 3, both the “sum” function runs fine and gives us the required output. The main thing to note here is that the name of the function can be the same but the data type and the sequence of arguments need to be different as shown in the example program otherwise program will not run.

Another example of function overloading is shown in Code Snippet 3.

```
// Calculate the volume of a cylinder
int volume(double r, int h){
    return(3.14 * r *r *h);
}

// Calculate the volume of a cube
int volume(int a){
    return (a * a * a);
}

// Rectangular box
int volume (int l, int b, int h){
    return (l*b*h);
}
```

Copy

Code Snippet 3: Volume Function Overloading Example

As shown in Code Snippet 3, we have created three “volume” functions, the 1st “volume” function calculates the volume of the cylinder and has two arguments “double r” and “int h”; the 2nd “volume” function calculates the volume of the cube and has one argument “int a”; the 3rd “volume” function calculates the volume of the rectangular box and has three arguments “int l”, “int b” and “int h”. The function call for these “volumes” function is shown in Code Snippet 4.

```
int main(){
    cout<<"The volume of cuboid of 3, 7 and 6 is "<<volume(3, 7, 6)<<endl;
    cout<<"The volume of cylinder of radius 3 and height 6 is "<<volume(3, 6)<<endl;
    cout<<"The volume of cube of side 3 is "<<volume(3)<<endl;
    return 0;
```

```
}
```

Copy

Code Snippet 4: Volume Function Call

As shown in Code Snippet 4, we passed three arguments in the first function call, two arguments in the second function call, and one argument in the third function call. The output of the following program is shown in figure 2.

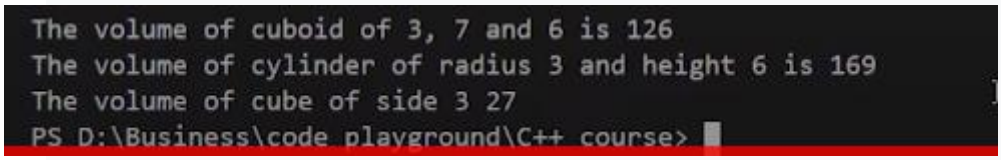


Figure 2: Volume Function Output

As shown in figure 2, all three “volume” functions run fine and give us the required output.

Code as described/written in the video

```
include<iostream>
using namespace std;

int sum(float a, int b){
    cout<<"Using function with 2 arguments"<<endl;
    return a+b;
}

int sum(int a, int b, int c){
    cout<<"Using function with 3 arguments"<<endl;
    return a+b+c;
}

// Calculate the volume of a cylinder
int volume(double r, int h){
    return(3.14 * r *r *h);
}

// Calculate the volume of a cube
int volume(int a){
    return (a * a * a);
}

// Rectangular box
int volume (int l, int b, int h){
    return (l*b*h);
}

int main(){
    cout<<"The sum of 3 and 6 is "<<sum(3,6)<<endl;
    cout<<"The sum of 3, 7 and 6 is "<<sum(3, 7, 6)<<endl;
    cout<<"The volume of cuboid of 3, 7 and 6 is "<<volume(3, 7, 6)<<endl;
    cout<<"The volume of cylinder of radius 3 and height 6 is "<<volume(3, 6)<<endl;
    cout<<"The volume of cube of side 3 is "<<volume(3)<<endl;
```

```
    return 0;
}
```

Object Oriented Programming in C++ | 20

In this series of our C++ tutorials, we will visualize object-oriented programming in the C++ language. In our last lecture, we discussed function overloading in C++.

Why Object-Oriented Programming?

Before we discuss object-oriented programming, we need to learn why we need object-oriented programming?

- C++ language was designed with the main intention of adding object-oriented programming to C language
- As the size of the program increases readability, maintainability, and bug-free nature of the program decrease.
- This was the major problem with languages like C which relied upon functions or procedure (hence the name procedural programming language)
- As a result, the possibility of not addressing the problem adequately was high
- Also, data was almost neglected, data security was easily compromised
- Using classes solves this problem by modeling program as a real-world scenario

Difference between Procedure Oriented Programming and Object-Oriented Programming

Procedure Oriented Programming

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

Object-Oriented Programming

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

Basic Concepts in Object-Oriented Programming

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

Benefits of Object-Oriented Programming

- Better code reusability using objects and inheritance
- Principle of data hiding helps build secure systems
- Multiple Objects can co-exist without any interference
- Software complexity can be easily managed

```
No Source Code Associated With This Video
```

Copy

Classes, Public and Private access modifiers in C++ | 21

In this tutorial, we will discuss classes, public and private access modifiers in C++

Why use classes instead of structures

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

Classes in C++

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

Public Access Modifier in C++

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

Private Access Modifier in C++

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

An example program to demonstrate classes, public and private access modifiers are shown in Code Snippet 1.

```
class Employee
{
    private:
        int a, b, c;
    public:
        int d, e;
        void setData(int a1, int b1, int c1); // Declaration
        void getData(){
            cout<<"The value of a is "<<a<<endl;
            cout<<"The value of b is "<<b<<endl;
            cout<<"The value of c is "<<c<<endl;
            cout<<"The value of d is "<<d<<endl;
            cout<<"The value of e is "<<e<<endl;
        }
};

void Employee :: setData(int a1, int b1, int c1){
    a = a1;
    b = b1;
    c = c1;
}
```

Copy

Code Snippet 1: Class Program

As shown in Code Snippet 1, 1st we created an “employee” class, 2nd three integer variables “int a”, “int b”, and “int c” were declared under the private access modifier, 3rd two integer variables “int d” and “int e” was declared under the public access modifiers, 4th “setData” function was declared, 5th “getData” function was defined and values of all the variables are printed. 6th “setData” function was defined outside the “employee” class by using a scope resolution operator; “setData” function is used to assign values to the private member of the class. An example to create the object of the class and use its class members is shown in Code Snippet 2.

```
int main(){
    Employee harry;
    harry.d = 34;
    harry.e = 89;
    harry.setData(1,2,4);
    harry.getData();
    return 0;
}
```

Copy

Code Snippet 2: Creating Object Example

As shown in Code Snippet 2, 1st we created an object “harry” of the class “employee”; 2nd we assigned values to “int d” and “int e” which are public class members. If we try to assign values to the private class member’s compiler will throw an error. 3rd we passed the values to the function “setData” and at the end, we called “getData” function which will print the values of all the variables. The output for the following program is shown in figure 1.

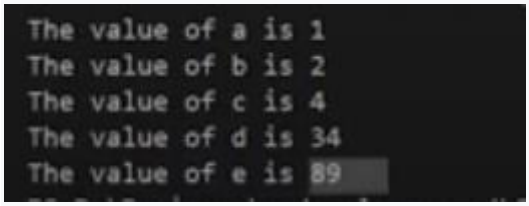


Figure 1: Class Program Output

As shown in figure 1, all the values of our data members are printed.

Code as described/written in the video

```
include<iostream>
using namespace std;

class Employee
{
    private:
        int a, b, c;
    public:
        int d, e;
        void setData(int a1, int b1, int c1); // Declaration
        void getData(){
            cout<<"The value of a is "<<a<<endl;
            cout<<"The value of b is "<<b<<endl;
            cout<<"The value of c is "<<c<<endl;
            cout<<"The value of d is "<<d<<endl;
            cout<<"The value of e is "<<e<<endl;
        }
};

void Employee :: setData(int a1, int b1, int c1){
    a = a1;
    b = b1;
    c = c1;
}
```

```
int main(){
    Employee harry;
    // harry.a = 134; -->This will throw error as a is private
    harry.d = 34;
    harry.e = 89;
    harry.setData(1,2,4);
    harry.getData();
    return 0;
}
```

Copy

OOPs Recap & Nesting of Member Functions in C++ | 22

In this tutorial, we will discuss the nesting of a member function in C++

Object-Oriented programming Recap

- Stroustrup initially named C++ language as C with classes because C++ language was almost the same as C language but they added a new concept of classes in it.
- Classes are the extension of structures in C language.
- Structures had limitations such as; members are public and no methods.
- Classes have some additional futures than structures such as; classes that can have methods and properties.
- Classes have a feature to make class members as public and private.
- In C++ objects can be declared along with class deceleration as shown in Code Snippet 1.

```
class Employee{
            // Class definition
} harry, rohan, lovish;
```

Copy

Code Snippet 1: Declaring Objects with Class Declaration

Nesting of Member Functions

If one member function is called inside the other member function of the same class it is called nesting of a member function. A program to demonstrate the nesting of a member function is shown below.

```
class binary
{
private:
    string s;
    void chk_bin(void);

public:
    void read(void);
    void ones_compliment(void);
    void display(void);
};
```

Copy

Code Snippet 2: Binary Class

As shown in Code Snippet 2, we created a binary class that has, “s” string variable and “chk_bin” void function as private class members; and “read” void function, “ones_compliment” void function, and “display” void function as public class members. The definitions of these functions are shown below.

```
void binary::read(void)
{
    cout << "Enter a binary number" << endl;
    cin >> s;
}
```

Copy

Code Snippet 3: Read Function

As shown in Code Snippet 3, we have created a “read” function. This function will take input from the user at runtime.

```
void binary::chk_bin(void)
{
    for (int i = 0; i < s.length(); i++)
    {
        if (s.at(i) != '0' && s.at(i) != '1')
        {
            cout << "Incorrect binary format" << endl;
            exit(0);
        }
    }
}
```

Copy

Code Snippet 4: Check Binary Function

As shown in Code Snippet 4 we have created a “chk_bin” function. This ”for” loop in the function will run till the length of the string and “if” condition in the body of the loop will check the whole string that if there are any values in the string other than “1” and “0”. If there are values other than “1” and “0” this function will output “Incorrect binary format”.

```
void binary::ones_compliment(void)
{
    chk_bin();
    for (int i = 0; i < s.length(); i++)
    {
        if (s.at(i) == '0')
        {
            s.at(i) = '1';
        }
       else
        {
            s.at(i) = '0';
        }
    }
}
```

Copy

Code Snippet 5: One's Compliment

As shown in Code Snippet 5, in the body of the “ones_compliment” function; the “chk_bin” function is called, and as we have discussed above that if one member function is called inside the other member function of the same class it is called **nesting of a member function**. The “for” loop inside

the “ones_compliment” functions runs till the length of the string and the “if” condition inside the loop replaces the number “0” with “1” and “1” with “0”.

```
void binary::display(void)
{
    cout<<"Displaying your binary number"<<endl;
    for (int i = 0; i < s.length(); i++)
    {
        cout << s.at(i);
    }
    cout<<endl;
}
```

Copy

Code Snippet 6: Display Function

As shown in Code Snippet 6, the “for” loop inside display function runs till the length of the string and prints each value of the sting.

```
int main()
{
    binary b;
    b.read();
    // b.chk_bin();
    b.display();
    b.ones_compliment();
    b.display();

    return 0;
}
```

Copy

Code Snippet 7: Main Function

As shown in Code Snippet 7, we created an object “b” of the binary data type, and the functions “read”, “display”, “ones_compliment”, and “display” are called. The main thing to note here is that the function ”chk_bin” is the private access modifier of the class so we cannot access it directly by using the object, it can be only accessed inside the class or by the member function of the class

Code as described/written in the video

```
// OOPs - Classes and objects


// C++ --> initially called --> C with classes by stroustroup
// class --> extension of structures (in C)
// structures had limitations
//      - members are public
//      - No methods
// classes --> structures + more
// classes --> can have methods and properties
// classes --> can make few members as private & few as public
// structures in C++ are typedefed
// you can declare objects along with the class declarion like this:
/* class Employee{
            // Class definition
```

```
        } harry, rohan, lovish; */
// harry.salary = 8 makes no sense if salary is private

// Nesting of member functions

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

class binary
{
private:
    string s;
    void chk_bin(void);

public:
    void read(void);
    void ones_compliment(void);
    void display(void);
};

void binary::read(void)
{
    cout << "Enter a binary number" << endl;
    cin >> s;
}

void binary::chk_bin(void)
{
    for (int i = 0; i < s.length(); i++)
    {
        if (s.at(i) != '0' && s.at(i) != '1')
        {
            cout << "Incorrect binary format" << endl;
            exit(0);
        }
    }
}

void binary::ones_compliment(void)
{
    chk_bin();
    for (int i = 0; i < s.length(); i++)
    {
        if (s.at(i) == '0')
        {
            s.at(i) = '1';
        }
       else
```

```
        {
            s.at(i) = '0';
        }
    }
}

void binary::display(void)
{
    cout<<"Displaying your binary number"<<endl;
    for (int i = 0; i < s.length(); i++)
    {
        cout << s.at(i);
    }
    cout<<endl;
}

int main()
{
    binary b;
    b.read();
    // b.chk_bin();
    b.display();
    b.ones_compliment();
    b.display();

    return 0;
}
```

C++ Objects Memory Allocation & using Arrays in Classes | 23

In this tutorial, we will discuss objects memory allocation and using arrays in C++

Objects Memory Allocation in C++

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

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

Arrays in Classes

Arrays are used to store multiple values of the same type. An array is very helpful when multiple variables are required, instead of making multiple variables one array can be used which can store multiple values. Array stores data in sequential order. An example program to demonstrate the use of arrays in classes is shown below.

```
class Shop
{
    int itemId[100];
    int itemPrice[100];
    int counter;
```

```
public:
    void initCounter(void) { counter = 0; }
    void setPrice(void);
    void displayPrice(void);
};
```

Copy

Code Snippet 1: Shop Class

As shown in Code Snippet 1, we created a shop class which has, “itemId[100]” and “itemPrice” as integer array variable and “counter” variable as private class members; and “initCounter” void function, “setPrice” void function, and “displayPrice” void function as public class members. The definitions of these functions are shown below.

```
void Shop ::setPrice(void)
{
    cout << "Enter Id of your item no " << counter + 1 << endl;
    cin >> itemId[counter];
    cout << "Enter Price of your item" << endl;
    cin >> itemPrice[counter];
    counter++;
}
```

Copy

Code Snippet 2: Set Price Function

As shown in Code Snippet 2, we have created a “setPrice” function. This function will take input for “itemId” and “ItemPrice” from the user at runtime. The value of the counter will be incremented by one every time this function will run.

```
void Shop ::displayPrice(void)
{
    for (int i = 0; i < counter; i++)
    {
        cout << "The Price of item with Id " << itemId[i] << " is " << itemPrice[i] << endl;
    }
}
```

Copy

Code Snippet 3: Display Price Function

As shown in Code Snippet 3, the “for” loop inside the “displayPrice” function runs till the length of the counter and prints values of the array “itemId” and “ItemPrice”.

```
int main()
{
    Shop dukaan;
    dukaan.initCounter();
    dukaan.setPrice();
    dukaan.setPrice();
    dukaan.setPrice();
    dukaan.displayPrice();
    return 0;
}
```

Copy

Code Snippet 4: Main Function

As shown in Code Snippet 4, we created an object “dukaan” of the shop data type, and the functions “initCounter” is called. The function “setPrice” is called three times. Loops can also be used to call the function multiple times. The “displayPrice” function is also called in the main function. The output of the following program is shown in figure 1.

```
Enter Id of your item no 1
1001
Enter Price of your item
12
Enter Id of your item no 2
1002
Enter Price of your item
23
Enter Id of your item no 3
1003
Enter Price of your item
34
The Price of item with Id 1001 is 12
The Price of item with Id 1002 is 23
The Price of item with Id 1003 is 34
```

Figure 1: Program Output

As shown in figure 1, for the item 1 we entered the ID “1001” and price “12”; for the item 2 we entered the ID “1002” and price “23”; for the item 3 we entered the ID “1003” and price “34”. The Output of the program has displayed the ID and the price of each item.

Code as described/written in the video

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

class Shop
{
    int itemId[100];
    int itemPrice[100];
    int counter;

public:
    void initCounter(void) { counter = 0; }
    void setPrice(void);
    void displayPrice(void);
};

void Shop ::setPrice(void)
{
    cout << "Enter Id of your item no " << counter + 1 << endl;
    cin >> itemId[counter];
    cout << "Enter Price of your item" << endl;
    cin >> itemPrice[counter];
    counter++;
}

void Shop ::displayPrice(void)
{
    for (int i = 0; i < counter; i++)
    {
```

```
        cout << "The Price of item with Id " << itemId[i] << " is " << itemPrice[i] << endl;
    }
}

int main()
{
    Shop dukaan;
    dukaan.initCounter();
    dukaan.setPrice();
    dukaan.setPrice();
    dukaan.setPrice();
    dukaan.displayPrice();
    return 0;
}
```

Copy

Static Data Members & Methods in C++ OOPS | 24

In this tutorial, we will discuss static data members and methods in C++

Static Data Members in C++

When a static data member is created, there is only a single copy of the data member which is shared between all the objects of the class. As we have discussed in our previous lecture that if the data members are not static then every object has an individual copy of the data member and it is not shared.

Static Methods in C++

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

```
class Employee
{
    int id;
    static int count;

public:
    void setData(void)
    {
        cout << "Enter the id" << endl;
        cin >> id;
        count++;
    }
    void getData(void)
    {
        cout << "The id of this employee is " << id << " and this is employee number " << count << endl;
    }

    static void getCount(void){
        // cout<<id; // throws an error
        cout<<"The value of count is "<<count<<endl;
    }
```

```
};
```

Copy

Code Snippet 1: Employee Class

As shown in Code Snippet 1, we created an employee class that has integer “id” variable and “count” static integer variable as private class members; and “setData” void function, “getData” void function, and “getCount” static void function as public class members. These functions are explained below.

We have defined a “setData” function. This function will take input for “id” from the user at runtime and increment in the count. The value of the counter will be incremented by one every time this function will run.

We have defined a “getData” function. This function will print the values of the variables “id” and “count”.

We have defined a static “getCount” function. This function will print the value of the variable count”. The main thing to note here is that “getCount” function is static, so if we try to access any data members or member functions which are not static the compiler will throw an error.

```
// Count is the static data member of class Employee
int Employee::count; // Default value is 0


int main()
{
    Employee harry, rohan, lovish;
    // harry.id = 1;
    // harry.count=1; // cannot do this as id and count are private


    harry.setData();
    harry.getData();
    Employee::getCount();


    rohan.setData();
    rohan.getData();
    Employee::getCount();


    lovish.setData();
    lovish.getData();
    Employee::getCount();


    return 0;
}
```

Copy

Code Snippet 2: main Program

As shown in Code Snippet 2:

- The count variable is declared whose default value is “0”.
- Then we created objects “harry”, “rohan”, and “lovish” of the employee data type
- The functions “setData”, “getData” are called by the object “harry”, the function “getCount” is called by using class name and scope resolution operator because it is a static method.
- The functions “setData”, “getData” are called by the object “rohan”, the function “getCount” is called by using class name and scope resolution operator because it is a static method.
- The functions “setData”, “getData” are called by the object “lovish”, the function “getCount” is called by using class name and scope resolution operator because it is a static method.

The output of the following program is shown in figures 1 and 2.

```
Enter the id
1
The id of this employee is 1 and this is employee number 1
The value of count is 1
```

Figure 1: Program Output 1

```
Enter the id
2
The id of this employee is 2 and this is employee number 2
The value of count is 2
Enter the id
3
The id of this employee is 3 and this is employee number 3
The value of count is 3
```

Figure 2: Program Output 2

As shown in figures 1 and 2, for the “harry” object we entered the ID “1”; for the “rohan” object we entered the ID “2”; and for the “lovish” object we entered the ID “3”. The Output of the program has displayed the ID and the count of each employee.

Code as described/written in the video

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

class Employee
{
    int id;
    static int count;

public:
    void setData(void)
    {
        cout << "Enter the id" << endl;
        cin >> id;
        count++;
    }
    void getData(void)
    {
        cout << "The id of this employee is " << id << " and this is employee number " << count << endl;
    }

    static void getCount(void){
        // cout<<id; // throws an error
        cout<<"The value of count is "<<count<<endl;
    }
};

// Count is the static data member of class Employee
int Employee::count; // Default value is 0
```

```
int main()
{
    Employee harry, rohan, lovish;
    // harry.id = 1;
    // harry.count=1; // cannot do this as id and count are private

    harry.setData();
    harry.getData();
    Employee::getCount();

    rohan.setData();
    rohan.getData();
    Employee::getCount();

    lovish.setData();
    lovish.getData();
    Employee::getCount();

    return 0;
}
```

Copy

Array of Objects & Passing Objects as Function Arguments in C++ | 25

In this tutorial, we will discuss an array of objects and passing objects as a function arguments in C++

An array of Objects in C++

An array of objects is declared the same as any other data-type array. An array of objects consists of class objects as its elements. If the array consists of class objects it is called an array of objects. An example program to demonstrate the concept of an array of objects is shown below.

```
class Employee
{
    int id;
    int salary;

public:
    void setId(void)
    {
        salary = 122;
        cout << "Enter the id of employee" << endl;
        cin >> id;
    }

    void getId(void)
    {
        cout << "The id of this employee is " << id << endl;
    }
};
```

Copy

Code Snippet 1: Employee Class

As shown in Code Snippet 1, we created an employee class that has integer “id” variable and “salary” integer variable as private class members; and “setId” void function, “getId” void function as public class members. These functions are explained below.

We have defined a “setId” function. In this function, the “salary” variable is assigned by the value “122” and the function will take input for “id” from the user at runtime. We have defined a “getId” function. This function will print the values of the variables “id”.

```
int main()
{
    Employee fb[4];
    for (int i = 0; i < 4; i++)
    {
        fb[i].setId();
        fb[i].getId();
    }

    return 0;
}
```

Copy

Code Snippet 2: main program

As shown in Code Snippet 2, we created an array “fb” of size “4” which is of employee data-type. The “for” loop is used to run “setId” and “getId” functions till the size of an array. The main thing to note here is that the objects can also be created individually but it is more convenient to use an array if too many objects are to be created. The output of the following program is shown in figure 1.

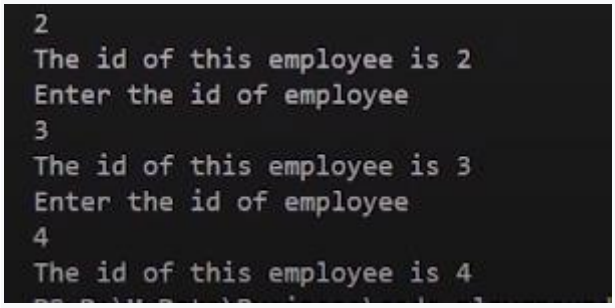


Figure 1: Employee Program Output

As shown in figure 1. As we input the Id for an employee it gives us the output of the employee Id.

Passing Object as Function Argument

Objects can be passed as function arguments. This is useful when we want to assign the values of a passed object to the current object. An example program to demonstrate the concept of passing an object as a function argument is shown below.

```
class complex{
    int a;
    int b;

    public:
        void setData(int v1, int v2){
            a = v1;
            b = v2;
        }
```

```
        void setDataBySum(complex o1, complex o2){
            a = o1.a + o2.a;
            b = o1.b + o2.b;
        }

        void printNumber(){
            cout<<"Your complex number is "<<a<<" + "<<b<<"i"<<endl;
        }
};
```

Copy

Code Snippet 3: Complex Class

As shown in Code Snippet 3, we created a complex class that has integer “a” variable and “b” integer variable as private class members; and “setData” void function, “setDataBySum” void function, and “printNumber” void function as public class members. These functions are explained below.

We have defined a “setData” function. In this function the values are assigned to the variables “a” and “b” because they are private data members of the class and values cannot be assigned directly. We have defined a “setDataBySum” function. In this function, the values of two objects are added and then assigned to the variables “a” and “b”. We have defined a “printNumber” function. In this function, the values of the variable “a” and “b” are being printed.

```
int main(){
    complex c1, c2, c3;
    c1.setData(1, 2);
    c1.printNumber();

    c2.setData(3, 4);
    c2.printNumber();

    c3.setDataBySum(c1, c2);
    c3.printNumber();
    return 0;
}
```

Copy

Code Snippet 4: main program 2

As shown in Code Snippet 4:

- We have created object “c1”, “c2”, and”c3” of complex data-type.
- The object “c1” calls the “setData” and “printNumber” functions.
- The object “c2” calls the “setData” and “printNumber” functions.
- The object “c3” calls the “setDataBySum” and “printNumber” functions.

The output of the following program is shown in figure 2.

```
Your complex number is 1+2i
Your complex number is 3+4i
Your complex number is 4+6i
```

Figure 2: Complex Program Output

Code as described/written in the video

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

```
class Employee
{
    int id;
    int salary;

public:
    void setId(void)
    {
        salary = 122;
        cout << "Enter the id of employee" << endl;
        cin >> id;
    }

    void getId(void)
    {
        cout << "The id of this employee is " << id << endl;
    }
};

int main()
{
    // Employee harry, rohan, lovish, shruti;
    // harry.setId();
    // harry.getId();
    Employee fb[4];
    for (int i = 0; i < 4; i++)
    {
        fb[i].setId();
        fb[i].getId();
    }

    return 0;
}
```

Copy

Code 25b as described/written in the video

```
include<iostream>
using namespace std;

class complex{
    int a;
    int b;

    public:
        void setData(int v1, int v2){
            a = v1;
            b = v2;
        }
```

```
        void setDataBySum(complex o1, complex o2){
            a = o1.a + o2.a;
            b = o1.b + o2.b;
        }

        void printNumber(){
            cout<<"Your complex number is "<<a<<" + "<<b<<"i"<<endl;
        }
};

int main(){
    complex c1, c2, c3;
    c1.setData(1, 2);
    c1.printNumber();

    c2.setData(3, 4);
    c2.printNumber();

    c3.setDataBySum(c1, c2);
    c3.printNumber();
    return 0;
}
```

Copy

Friend Functions in C++ | 26

In this tutorial, we will discuss friend function in C++

Friend Function in C++

Friend functions are those functions that have the right to access the private data members of class even though they are not defined inside the class. It is necessary to write the prototype of the friend function. One main thing to note here is that if we have written the prototype for the friend function in the class it will not make that function a member of the class. An example program to demonstrate the concept of friend function is shown below.

```
class Complex{
    int a, b;
    friend Complex sumComplex(Complex o1, Complex o2);
    public:
        void setNumber(int n1, int n2){
            a = n1;
            b = n2;
        }

        // Below line means that non member - sumComplex funtion is allowed to do anything with my private parts (members)
        void printNumber(){
            cout<<"Your number is "<<a<<" + "<<b<<"i"<<endl;
        }
};

Complex sumComplex(Complex o1, Complex o2){
```

```
    Complex o3;
    o3.setNumber((o1.a + o2.a), (o1.b+o2.b))
    ;
    return o3;
}
```

Copy

Code Snippet 1: Complex Class

As shown in Code Snippet 1, we created a complex class that has integer “a” variable and “b” integer variable as private class members; and “setNumber” void function, “printNumber” void function as public class members. The “sumComplex” friend function prototype is written as well in the complex class. These functions are explained below.

We have defined a “setNumber” function. In this function the values are assigned to the variables “a” and “b” because they are private data members of the class and values cannot be assigned directly. We have defined a “printNumber” function. In this function, the values of the variable “a” and “b” are being printed. We have defined a “sumComplex” friend function. In this function, the object “o3” is created which calls the “setNumber” function and passes the values of two objects after performing addition on them.

```
int main(){
    Complex c1, c2, sum;
    c1.setNumber(1, 4);
    c1.printNumber();

    c2.setNumber(5, 8);
    c2.printNumber();

    sum = sumComplex(c1, c2);
    sum.printNumber();

    return 0;
}
```

Copy

Code Snippet 2: main Program

As shown in Code Snippet 2:

- We have created object “c1”, “c2”, and”sum” of complex data-type.
- The object “c1” calls the “setNumber” and “printNumber” functions.
- The object “c2” calls the “setNumber” and “printNumber” functions.
- The function “sumComplex” is called and the values are assigned to the “sum”.
- The object “sum” calls the “printNumber” functions.

The output of the following program is shown in figure 1.

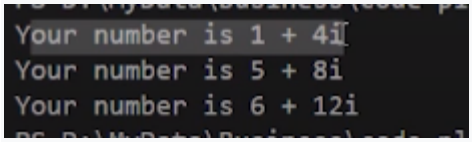


Figure 1: Complex Program Output

As shown in figure 1, the output of the complex number program is printed.

Properties of Friend Function

- Not in the scope of the class
- Since it is not in the scope of the class, it cannot be called from the object of that class, for example, **sumComplex()** is invalid
- A friend function can be invoked without the help of any object
- Usually contain objects as arguments
- Can be declared under the public or private access modifier, it will not make any difference
- It cannot access the members directly by their names, it needs (object_name.member_name) to access any member.

Code as described/written in the video

```
include<iostream>
using namespace std;


// 1 + 4i
// 5 + 8i
// -------
// 6 + 12i
class Complex{
    int a, b;
    friend Complex sumComplex(Complex o1, Complex o2);
    public:
        void setNumber(int n1, int n2){
            a = n1;
            b = n2;
        }

        // Below line means that non member - sumComplex funtion is allowed to do anything with my private parts (members)
        void printNumber(){
            cout<<"Your number is "<<a<<" + "<<b<<"i"<<endl;
        }
};

Complex sumComplex(Complex o1, Complex o2){
    Complex o3;
    o3.setNumber((o1.a + o2.a), (o1.b+o2.b))
    ;
    return o3;
}

int main(){
    Complex c1, c2, sum;
    c1.setNumber(1, 4);
    c1.printNumber();

    c2.setNumber(5, 8);
    c2.printNumber();

    sum = sumComplex(c1, c2);
    sum.printNumber();

    return 0;
}
```

```
/* Properties of friend functions
1. Not in the scope of class
2. since it is not in the scope of the class, it cannot be called from the object of that class. c1.sumComplex() == Invalid
3. Can be invoked without the help of any object
4. Usually contans the objects as arguments
5. Can be declared inside public or private section of the class
6. It cannot access the members directly by their names and need object_name.member_name to access any member.

*/
```

Friend Classes & Member Friend Functions in C++ | 27

In this tutorial, we will discuss friend classes and member friend functions in C++

Member Friend Functions in C++

Friend functions are those functions that have the access to private members of the class in which they are declared. The main thing to note here is that only that function can access the member function which is made a friend of the other class. An example of the friend function is shown below.

```
class Complex
{
    int a, b;
    // Individually declaring functions as friends
     friend int Calculator ::sumRealComplex(Complex, Complex);
     friend int Calculator ::sumCompComplex(Complex, Complex);

public:
    void setNumber(int n1, int n2)
    {
        a = n1;
        b = n2;a
    }

    void printNumber()
    {
        cout << "Your number is " << a << " + " << b << "i" << endl;
    }
};

int Calculator ::sumRealComplex(Complex o1, Complex o2)
{
    return (o1.a + o2.a);
}

int Calculator ::sumCompComplex(Complex o1, Complex o2)
{
    return (o1.b + o2.b);
}
```

Copy

Code Snippet 1: Friend function example

As shown in a code snippet 1, a complex class is created which consists of two friend functions “sumRealComplex” and “sumCompComplex” of the calculator class. The main thing to note here is that “sumRealComplex” and “sumCompComplex” are the friend functions of complex class so they can access all the private members of the complex class.

Friend Classes in C++

Friend classes are those classes that have permission to access private members of the class in which they are declared. The main thing to note here is that if the class is made friend of another class then it can access all the private members of that class. An example program to demonstrate friend classes in C++ is shown below.

```
// Forward declaration
class Complex;


class Calculator
{
public:
    int add(int a, int b)
    {
        return (a + b);
    }

    int sumRealComplex(Complex, Complex);
    int sumCompComplex(Complex, Complex);
};
```

Copy

Code Snippet 2: Calculator Class

As shown in code snippet 2, a complex class is declared at the top which is known as forward declaration. Forward declaration hints to the compiler that this class is declared somewhere forward in the code. After that calculator class is defined this consists of three public member functions, “add”, “sumRealComplex”, and “sumCompComplex”. The “add” function will add the values of “a” and “b” and return the value. The “sumRealComplex” and “sumCompComplex” are taking two objects of the complex class. The code for the complex class is shown below.

```
class Complex
{
    int a, b;
    // Individually declaring functions as friends
    // friend int Calculator ::sumRealComplex(Complex, Complex);
    // friend int Calculator ::sumCompComplex(Complex, Complex);


    // Aliter: Declaring the entire calculator class as friend
    friend class Calculator;


public:
    void setNumber(int n1, int n2)
    {
        a = n1;
        b = n2;a
    }


    void printNumber()
    {
        cout << "Your number is " << a << " + " << b << "i" << endl;
    }
};
```

```
int Calculator ::sumRealComplex(Complex o1, Complex o2)
{
    return (o1.a + o2.a);
}

int Calculator ::sumCompComplex(Complex o1, Complex o2)
{
    return (o1.b + o2.b);
}
```

Copy

Code Snippet 3: Complex Class

As shown in code snippet 3, a complex class is defined which consists of, two private data members “a” and “b”, and two public member functions “setNumber” and “printNumber”. The function “setNumber” will assign the values to the variables “a” and “b”. The function “printNumber” will print the values of the variables “a” and “b”. Two functions “sumRealComplex” and “sumCompComplex” are defined at the end. The function “sumRealComplex” will add the real values and the function “sumCompComplex” will add the complex value. The main program is shown below.

```
int main()
{
    Complex o1, o2;
    o1.setNumber(1, 4);
    o2.setNumber(5, 7);
    Calculator calc;
    int res = calc.sumRealComplex(o1, o2);
    cout << "The sum of real part of o1 and o2 is " << res << endl;
    int resc = calc.sumCompComplex(o1, o2);
    cout << "The sum of complex part of o1 and o2 is " << resc << endl;
    return 0;
}
```

Copy

Code snippet 4: Main Program

As shown in code snippet 4, 1st two objects “o1” and “o2” of the “complex” data type are declared. 2nd “setNumber” function is called with the “o1” and “o2” objects and the values are passed. 3rd object “calc” of the calculator data type is declared. 4th “sumRealComplex” function is called by the “calc” object and the object “o1” and “o2” are passed to it. 5th “sumCompComplex” function is called by the “calc” object and the object “o1” and “o2” are passed to it. The output of the following program is shown in figure 1.

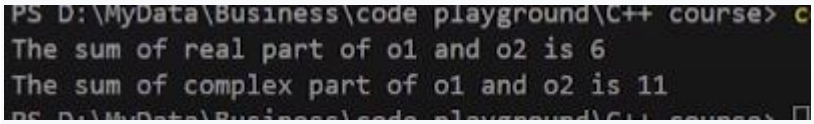


Figure 1: Program Output

As shown in figure 1, the sum of the real part is shown which is “6” and the sum of the complex part is shown which is “11”.

More on C++ Friend Functions (Examples & Explanation) | 28

In this tutorial, we will discuss more on friend functions in C++ with examples

Friend Functions in C++

As we have already discussed in previous lectures friend functions are those functions that can access the private data members of the other class. An example program to demonstrate friend functions in C++ is shown below.

Friend Function Example 1

```
class Y;


class X{
    int data;
    public:
        void setValue(int value){
            data = value;
        }
    friend void add(X, Y);
};

class Y{
    int num;
    public:
        void setValue(int value){
            num = value;
        }
    friend void add(X, Y);


};


void add(X o1, Y o2){
    cout<<"Summing data of X and Y objects gives me "<< o1.data + o2.num;
}
```

Copy

Code Snippet 1: Friend Function Example 1

As shown in a code snippet 1,

- 1st class “Y” is declared at the top which is known as forward declaration to let the compiler know that this class is defined somewhere in the program.
- 2nd class “X” is defined which consists of private data member “data” and public member function “setValue” which assigns the value to the private data member “data”. At the end friend function “add” is declared.
- 3rd class “Y” is defined which consists of private data member “num” and public member function “setValue” which assigns the value to the private data member “num”. At the end friend function “add” is declared.
- 4th function “add” is defined which add the value of the objects of class “X” and “Y” and print it.

The main program is shown in Code Snippet 2.

```
int main(){
    X a1;
    a1.setValue(3);


    Y b1;
    b1.setValue(15);


    add(a1, b1);
    return 0;
}
```

Copy

Code Snippet 2: Main Program

As shown in Code Snippet 2,

- 1st object “a1” of the data type “X” is declared
- 2nd function “setValue” is called by the object “a1” and the value “3” is passed
- 3rd object “b1” of the data type “Y” is declared
- 4th function “setValue” is called by the object “b1” and the value “15” is passed
- 5th function “add” is called and the objects “a1” and “b1” are passed to it. The function “add” will add the values of both objects and print them.

The output of the following program is shown in figure 1.

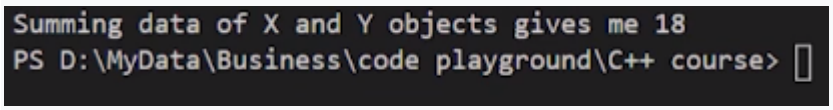


Figure 1: Program Output 1

As shown in figure 1, the sum of both values is shown which is “18”.

Friend Function Example 2

```
class c2;

class c1{
    int val1;
    friend void exchange(c1 & , c2 &);
    public:
        void indata(int a){
            val1 = a;
        }

        void display(void){
            cout<< val1 <<endl;
        }
};

class c2{
    int val2;
    friend void exchange(c1 &, c2 &);
    public:
        void indata(int a){
            val2 = a;
        }

        void display(void){
            cout<< val2 <<endl;
        }
};

void exchange(c1 &x, c2 &y){
    int tmp = x.val1;
    x.val1 = y.val2;
```

```
    y.val2 = tmp;
}
```

Copy

Code Snippet 3: Friend Function Example 2

As shown in a code snippet 3,

- 1st class “c2” is declared at the top which is known as forward declaration to let the compiler know that this class is defined somewhere in the program.
- 2nd class “c1” is defined which consists of private data member “val1” and friend function “exchange” which takes reference variables “c1” and “c2” as parameters. The public member function “indata” is defined which assigns the value to the private data member “val1” and the function “display” prints the value of the data member “val1”.
- 3rd class “c2” is defined which consists of private data member “val2” and friend function “exchange” which takes reference variables “c1” and “c2” as parameters. The public member function “indata” is defined which assigns the value to the private data member “val2” and the function “display” prints the value of the data member “val2”.
- 4th function “exchange” is defined which swap the values.

The main program is shown in Code Snippet 4.

```
int main(){
    c1 oc1;
    c2 oc2;

    oc1.indata(34);
    oc2.indata(67);
    exchange(oc1, oc2);

    cout<<"The value of c1 after exchanging becomes: ";
    oc1.display();
    cout<<"The value of c2 after exchanging becomes: ";
    oc2.display();

    return 0;
}
```

Copy

Code Snippet 4: Main program

As shown in Code Snippet 4,

- 1st object “oc1” of the data type “c1” is declared
- 2nd object “oc2” of the data type “c2” is declared
- 3rd function “indata” is called by the object “oc1” and the value “34” is passed
- 4th function “indata” is called by the object “oc2” and the value “67” is passed
- 5th function “exchange” is called and the objects “oc1” and “oc2” are passed to it. The function “exchange” will swap both values and
- 6th function “display” is called by the objects “oc1” and “oc2” which will print their values.

The output of the following program is shown in figure 2.

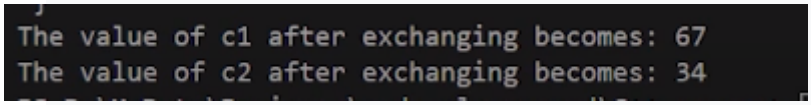


Figure 2: Program Output 2

As shown in figure 2, the values are swapped.

Constructors In C++ | 29

In this tutorial, we will discuss constructors in C++

Constructors in C++

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

Important Characteristics of Constructors in C++

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

An example program to demonstrate the concept of the constructor is shown below.

```
include <iostream>
using namespace std;

class Complex
{
    int a, b;

public:
    // Creating a Constructor
    // Constructor is a special member function with the same name as of the class.
    //It is used to initialize the objects of its class
    //It is automatically invoked whenever an object is created

    Complex(void); // Constructor declaration

    void printNumber()
    {
        cout << "Your number is " << a << " + " << b << "i" << endl;
    }
};

Complex ::Complex(void) // ----> This is a default constructor as it takes no parameters
{
    a = 10;
    b = 0;
    // cout<<"Hello world";
}
```

Copy

Code Snippet 1: Constructor Example Program

As shown in a code snippet 1,

- 1st "complex" class is defined which consists of private data members "a" and "b".
- 2nd default constructor of the "complex" class is declared.
- 3rd function "printNumber" is defined which will print the values of the data members "a" and "b".
- 4th default constructor is defined which will assign the values to the data members "a" and "b". The main things to note here are that whenever a new object will be created this constructor will run and if the parameters are not passed to the constructor it is called a default constructor.

The main program is shown in code snippet 2.

```
int main()
{
    Complex c1, c2, c3;
    c1.printNumber();
    c2.printNumber();
    c3.printNumber();

    return 0;
}
```

Copy

Code Snippet 2: Main Program

As shown in Code Snippet 2,

- 1st objects "c1", "c2", and "c3" of the complex data type are created. The main thing to note here is that when we are creating objects the constructor will run for each object and will assign the values.
- 2nd function "printNumber" is called by the objects "c1", "c2", and "c3".

The output for the following program is shown in figure 1.

```
Your number is 10 + 0i
Your number is 10 + 0i
Your number is 10 + 0i
PS D:\MyData\Business\code playground\C++ course>
```

Figure 1: Program Output

As shown in figure 1, whenever a "printNumber" function is called it prints the values which are being assigned through the constructor.

Parameterized and Default Constructors In C++ | 30

In this tutorial, we will discuss parameterized and default constructors in C++

Parameterized and Default Constructors in C++

Parameterized constructors are those constructors that take one or more parameters. Default constructors are those constructors that take no parameters. The main things to note here are that constructors are written in the public section of the class and the constructors don't have a return type. An example program to demonstrate the concept of the constructor is shown below.

Parameterized Constructors Example Program 1

```
include<iostream>
using namespace std;


class Complex
```

```
{
    int a, b;

public:
    Complex(int, int); // Constructor declaration

    void printNumber()
    {
        cout << "Your number is " << a << " + " << b << "i" << endl;
    }
};

Complex ::Complex(int x, int y) // ----> This is a parameterized constructor as it takes 2 parameters
{
    a = x;
    b = y;
    // cout<<"Hello world";
}
```

Copy

Code Snippet 1: Parameterized Constructor Example Program 1

As shown in a code snippet 1,

- 1st "complex" class is defined which consists of private data members "a" and "b".
- 2nd parameterized constructor of the "complex" class is declared which takes two parameters.
- 3rd function "printNumber" is defined which will print the values of the data members "a" and "b".
- 4th parameterized constructor is defined which takes two parameters and assigns the values to the data members "a" and "b". The main things to note here are that whenever a new object will be created this constructor will run.

The main program is shown in code snippet 2.

```
int main(){
    // Implicit call
    Complex a(4, 6);
    a.printNumber();

    // Explicit call
    Complex b = Complex(5, 7);
    b.printNumber();

    return 0;
}
```

Copy

Code Snippet 2: Main Program

As shown in Code Snippet 2,

- 1st parameterized constructor is called implicitly with the object "a" and the values "4" and "6" are passed
- 2nd function "printNumber" is called which will print the values of data members
- 3rd parameterized constructor is called explicitly with the object "b" and the values "5" and "7" are passed
- 4th function "printNumber" is called again which will print the values of data members

The output for the following program is shown in figure 1.

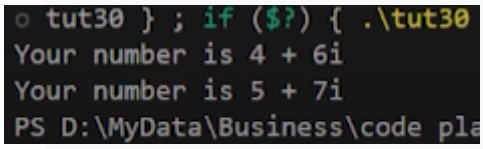


Figure 1: Program Output 1

Parameterized Constructors Example Program 2

```
include<iostream>
using namespace std;


class Point{
    int x, y;
    public:
        Point(int a, int b){
            x = a;
            y = b;
        }

        void displayPoint(){
            cout<<"The point is ("<<x<<", "<<y<<")"<<endl;
        }


};
```

Copy

Code Snippet 3: Parameterized Constructor Example Program 2

As shown in Code Snippet 3,

- 1st "point" class is defined which consists of private data members "x" and "y".
- 2nd parameterized constructor of the "point" class is defined which takes two parameters and assigns the values to the private data members of the class.
- 3rd function "displayPoint" is defined which will print the values of the data members "x" and "y".

The main program is shown in code snippet 4.

```
int main(){
    Point p(1, 1);
    p.displayPoint();


    Point q(4, 6);
    q.displayPoint();
    return 0;
}
```

Copy

Code Snippet 4: Main Program

As shown in Code Snippet 4,

- 1st parameterized constructor is called implicitly with the object “p” and the values “1” and “1” are passed
- 2nd function “displayPoint” is called which will print the values of data members
- 3rd parameterized constructor is called implicitly with the object “q” and the values “4” and “6” are passed
- 4th function “displayPoint” is called which will print the values of data members

The output for the following program is shown in figure 2.

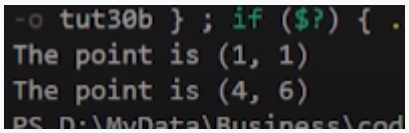


Figure 2: Program Output 2

Constructor Overloading In C++ | 31

In this tutorial, we will discuss constructor overloading in C++

Constructor Overloading in C++

Constructor overloading is a concept in which one class can have multiple constructors with different parameters. The main thing to note here is that the constructors will run according to the arguments for example if a program consists of 3 constructors with 0, 1, and 2 arguments, so if we pass 1 argument to the constructor the compiler will automatically run the constructor which is taking 1 argument. An example program to demonstrate the concept of Constructor overloading in C++ is shown below.

```
include <iostream>
using namespace std;

class Complex
{
    int a, b;

public:
    Complex(){
        a = 0;
        b =0;
    }

    Complex(int x, int y)
    {
        a = x;
        b = y;
    }

    Complex(int x){
        a = x;
        b = 0;
    }




    void printNumber()
```

```
    {
        cout << "Your number is " << a << " + " << b << "i" << endl;
    }
};
```

Copy

Code Snippet 1: Constructor Overloading Program Example

As shown in Code Snippet 1,

- 1st we created a “complex” class which consists of private data members “a” and “b”.
- 2nd default constructor of the “complex” class is declared which has no parameters and assigns “0” to the data members “a” and “b”.
- 3rd parameterized constructor of the “complex” class is declared which takes two parameters and assigns values to the data members “a” and “b”.
- 4th parameterized constructor of the “complex” class is declared which takes one parameter and assigns values to the data members “a” and “b”.
- 5th function “printNumber” is defined which will print the values of the data members “a” and “b”.

The main program is shown in code snippet 2.

```
int main()
{
    Complex c1(4, 6);
    c1.printNumber();

    Complex c2(5);
    c2.printNumber();

    Complex c3;
    c3.printNumber();
    return 0;
}
```

Copy

Code Snippet 2: Main Program

As shown in Code Snippet 2,

- 1st parameterized constructor is called with the object “c1” and the values “4” and “6” are passed. The main thing to note here is that this will run the constructor with two parameters.
- 2nd function “printNumber” is called which will print the values of data members
- 3rd parameterized constructor is called with the object “c2” and the value “5” is passed. The main thing to note here is that this will run the constructor with one parameter.
- 4th function “printNumber” is called which will print the values of data members
- 5th default constructor is called with the object “c3”. The main thing to note here is that this will run the constructor with no parameters.
- 6th function “printNumber” is called which will print the values of data members

The output for the following program is shown in figure 1.

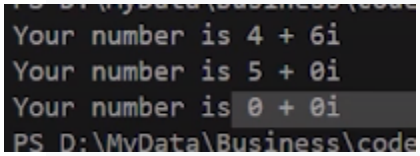


Figure 1: Program Output

As shown in figure 1, all the values which were passed and assigned through parameterized constructors and the values which were assigned through the default constructor are printed.

Constructors With Default Arguments In C++ | 32

In this tutorial, we will discuss constructors with default arguments in C++

Constructors with Default Arguments in C++

Default arguments of the constructor are those which are provided in the constructor declaration. If the values are not provided when calling the constructor the constructor uses the default arguments automatically. An example program to demonstrate the concept default arguments in C++ is shown below.

```
include<iostream>
using namespace std;

class Simple{
    int data1;
    int data2;
    int data3;

    public:
        Simple(int a, int b=9, int c=8){
            data1 = a;
            data2 = b;
            data3 = c;
        }

        void printData();

};

void Simple :: printData(){
    cout<<"The value of data1, data2 and data3 is "<<data1<<", "<< data2<<" and "<< data3<<endl;
}
```

Copy

Code Snippet 1: Constructor with Default Arguments Program Example

As shown in a code snippet 1,

- 1st we created a “simple” class which consists of private data members “data1”, “data2” and “data3”.
- 2nd parameterized constructor of the “simple” class is defined which takes three parameters and assigns values to the data members “a” and “b”. The main thing to note here is that the value “9” and “8” are the default values for the variables “b” and “c”.
- 3rd function “printData” is defined which prints the values of the data members “data1”, “data2”, and “data3”.

The main program is shown in code snippet 2.

```
int main(){
    Simple s(12, 13);
    s.printData();
    return 0;
}
```

Copy

Code Snippet 2: Main Program

As shown in code snippet 2,

- 1st parameterized constructor is called with the object “s” of the data type “simple” and the values “12” and “13” are passed. The main thing to note here is that the value of the parameter “c” will be automatically set by the default value.
- 2nd function “printData” is called which will print the values of data members.

The output for the following program is shown in figure 1.

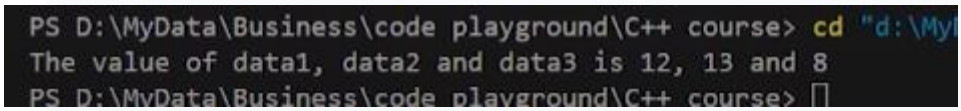


Figure 1: Program Output

As shown in figure 1, the value “12”, “13”, and “8” are printed. The constructor assigned the values “12” and “13” to the variables “a” and “b” but the value for the variable “c” was not passed that’s why constructors set the value “8” which was the default value for the variable “c”.

Dynamic Initialization of Objects Using Constructors | 33

In this tutorial, we will discuss the dynamic initialization of objects using constructors in C++

Dynamic Initialization of Objects Using Constructors

The dynamic initialization of the object means that the object is initialized at the runtime. Dynamic initialization of the object using a constructor is beneficial when the data is of different formats. An example program is shown below to demonstrate the concept of dynamic initialization of objects using constructors.

```
include<iostream>
using namespace std;



class BankDeposit{
    int principal;
    int years;
    float interestRate;
    float returnValue;

    public:
        BankDeposit(){}
        BankDeposit(int p, int y, float r); // r can be a value like 0.04
        BankDeposit(int p, int y, int r); // r can be a value like 14
        void show();
};
```

Copy

Code Snippet 1: Dynamic Initialization of Objects using Constructor Example

As shown in Code Snippet 1,

- 1st we created a “BankDeposit” class which consists of private data members “principal”, “years”, “interestRate”, and “returnValue”.
- 2nd default constructor of the “BankDeposit” class is declared.
- 3rd parameterized constructor of the “BankDeposit” class is declared which takes three parameters “p”, “y”, and “r”. The main thing to note here is that the parameter “r” is of a float data type.
- 4th parameterized constructor of the “BankDeposit” class is declared which takes three parameters “p”, “y”, and “r”. The main thing to note here is that the parameter “r” is of an integer data type.

- 5th function “show” is declared.

The definition of constructors and function is shown below.

```
BankDeposit :: BankDeposit(int p, int y, float r)
{
    principal = p;
    years = y;
    interestRate = r;
    returnValue = principal;
    for (int i = 0; i < y; i++)
    {
        returnValue = returnValue * (1+interestRate);
    }
}

BankDeposit :: BankDeposit(int p, int y, int r)
{
    principal = p;
    years = y;
    interestRate = float(r)/100;
    returnValue = principal;
    for (int i = 0; i < y; i++)
    {
        returnValue = returnValue * (1+interestRate);
    }
}

void BankDeposit :: show(){
    cout<<endl<<"Principal amount was "<<principal
        << ". Return value after "<<years
        << " years is "<<returnValue<<endl;
}
```

Copy

Code Snippet 2: Definition of Constructors and Function

As shown in Code snippet 2,

- 1st the constructor “BankDeposit” is defined in which the value of the parameter “p” is assigned to the data member “principal”; the value of the parameter “y” is assigned to the data member “year”; the value of the parameter “r” is assigned to the data member “interestRate”. At the end “for” loop is defined which will run till the length of the variable “y” and add “1” in the “interestRate”; then multiply the value with the “returnValue”. The main thing to note here is that in this constructor the data type of the parameter “r” is float.
- 2nd another constructor “BankDeposit” is defined in which the value of the parameter “p” is assigned to the data member “principal”; the value of the parameter “y” is assigned to the data member “year”; the value of the parameter “r” is converted to “float” and divided by “100” then assigned to the data member “interestRate”. At the end “for” loop is defined which will run till the length of the variable “y” and add “1” in the “interestRate”; then multiply the value with the “returnValue”. The main thing to note here is that in this constructor the data type of the parameter “r” is float.
- 3rd the function “show” is defined which will print the values of the data members “principal”, “year”, and “returnValue”.

The main program is shown in code snippet 3.

```
int main(){
    BankDeposit bd1, bd2, bd3;
    int p, y;
```

```
    float r;
    int R;



    cout<<"Enter the value of p y and r"<<endl;
    cin>>p>>y>>r;
    bd1 = BankDeposit(p, y, r);
    bd1.show();

    cout<<"Enter the value of p y and R"<<endl;
    cin>>p>>y>>R;
    bd2 = BankDeposit(p, y, R);
    bd2.show();
    return 0;
}
```

Copy

Code Snippet 3: Main Program

As shown in a code snippet 3,

- 1st the object “bd1”, “bd2”, and “bd3” of the data type “BankDeposit” are created.
- 2nd the integer variables “p” and “y” are declared; the float variable “r” is declared, and the integer variable “R” is declared.
- 3rd the values for the variables “p”, “y”, and”r” are taken from the user on the runtime.
- 4th parameterized constructor “BankDeposit” is called with the object “bd1” and the variables “p”, “y”, and “r” are passed. The main thing to note here is that this will run the constructor with float parameters “r”.
- 5th function “show” is called which will print the values of data members
- 6th the values for the variables “p”, “y”, and ”R” are taken from the user on the runtime.
- 7th parameterized constructor “BankDeposit” is called with the object “bd2” and the variables “p”, “y”, and “R” are passed. The main thing to note here is that this will run the constructor with integer parameters “R”.
- 8th function “show” is called which will print the values of data members.

The output for the following program is shown in figure 1.

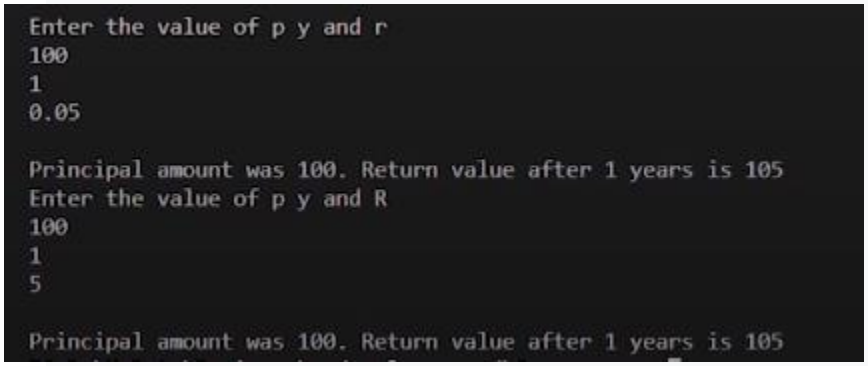


Figure 1: Program Output

As shown in figure 1, the first time the values “100”, “1”, and “0.05” are entered and it gives us the return value of “105”. The second time the values “100”, “1”, and “5” are entered and it gives us the return value of “105”. So the main thing to note here is that the compiler figures out the run time by seeing the data type and runs the relevant constructor.

Copy Constructor in C++ | 34

In this tutorial, we will discuss copy constructor in C++

Copy Constructor in C++

A copy constructor is a type of constructor that creates a copy of another object. If we want one object to resemble another object we can use a copy constructor. If no copy constructor is written in the program compiler will supply its own copy constructor. An example program to demonstrate the concept of a Copy constructor in C++ is shown below.

```
include<iostream>
using namespace std;



class Number{
    int a;
    public:
        Number(){
            a = 0;
        }

        Number(int num){
            a = num;
        }
        // When no copy constructor is found, compiler supplies its own copy constructor
        Number(Number &obj){
            cout<<"Copy constructor called!!!"<<endl;
            a = obj.a;
        }

        void display(){
            cout<<"The number for this object is "<< a <<endl;
        }
};
```

Copy

Code Snippet 1: Copy Constructor Example Program

As shown in Code Snippet 1,

- 1st we created a “number” class which consists of private data member “a”.
- 2nd default constructor of the “number” class is defined which has no parameters and assign “0” to the data members “a”.
- 3rd parameterized constructor of the “number” class is defined which takes one parameter and assigns values to the data members “a”.
- 4th copy constructor of the “number” class is defined which takes its own reference object as a parameter and assigns values to the data members “a”.
- 5th function “display” is defined which will print the values of the data members “a”.

The main program is shown in code snippet 2.

```
int main(){
    Number x, y, z(45), z2;
    x.display();
    y.display();
    z.display();

    Number z1(z); // Copy constructor invoked
    z1.display();
```

```
    z2 = z; // Copy constructor not called
    z2.display();


    Number z3 = z; // Copy constructor invoked
    z3.display();


    // z1 should exactly resemble z  or x or y


    return 0;
}
```

Copy

Code Snippet 2: Main Program

As shown in Code Snippet 2,

- 1st objects “x”, “y”, “z”, and “z1” are created of the “number” data type. The main thing to note here is that the object “z” has a value “45”.
- 2nd function “display” is called by the objects “x”, “y”, and “z”.
- 3rd copy constructor is invoked and the object “z” is passed to “z1”
- 4th function “display” is called by the object “z1”
- 5th the value of “z” is assigned to “z1”. The main thing to note here is that it will not invoke a copy constructor because the object “z” is already created.
- 6th function “display” is called by the object “z2”
- 7th the value of “z” is assigned to “z3”. The main thing to note here is that it will invoke a copy constructor because the object “z3” is being created.
- 8th function “display” is called by the object “z3”

The output for the following program is shown in figure 1.

```
PS D:\MyData\Business\code playground\C
The number for this object is 0
The number for this object is 0
The number for this object is 45
Copy constructor called!!!
The number for this object is 45
The number for this object is 45
Copy constructor called!!!
```

Figure 1: Program Output

As shown in figure 1, all the values which were passed and assigned through copy constructors are printed.

Destructor in C++ in Hindi | 35

In this tutorial, we will discuss Destructor in C++

Destructor in C++

A destructor is a type of function which is called when the object is destroyed. Destructor never takes an argument nor does it return any value. An example program to demonstrate the concept of destructors in C++ is shown below.

```
include<iostream>
using namespace std;


// Destructor never takes an argument nor does it return any value
int count=0;
```

```
class num{
    public:
        num(){
            count++;
            cout<<"This is the time when constructor is called for object number"<<count<<endl;
        }

        ~num(){
            cout<<"This is the time when my destructor is called for object number"<<count<<endl;
            count--;
        }
};
```

Copy

Code Snippet 1: Destructor Example Program

As shown in Code Snippet 1,

- 1st global variable “count” is initialized.
- 2nd we created a “num” class.
- 3rd default constructor of the “num” class is defined which has no parameters and does increment in the variable “count” and prints its value. The main thing to note here is that every time the new object will be created this constructor will run.
- 4th destructor of the “num” class is defined. The destructor prints the value of the variable “count” and decrement in the value of “count”. The main thing to note here is that every time the object has been destroyed this destructor will run.

The main program is shown in code snippet 2.

```
int main(){
    cout<<"We are inside our main function"<<endl;
    cout<<"Creating first object n1"<<endl;
    num n1;
    {
        cout<<"Entering this block"<<endl;
        cout<<"Creating two more objects"<<endl;
        num n2, n3;
        cout<<"Exiting this block"<<endl;
    }
    cout<<"Back to main"<<endl;
    return 0;
}
```

Copy

Code Snippet 2: Main Program

As shown in Code Snippet 2,

- 1st object “n1” is created of the “num” data type. The main thing to note here is that when the object “n1” is created the constructor will run.
- 2nd inside the block two objects “n2” and “n3” are created of the “num” data type. The main things to note here are that when the objects “n2” and “n3” are created the constructor will run for both objects and when the block ends the destructor will run for both objects “n2” and “n3”.
- 3rd when the program ends the destructor for the object “n1” will run.

The output for the following program is shown in figure 1.

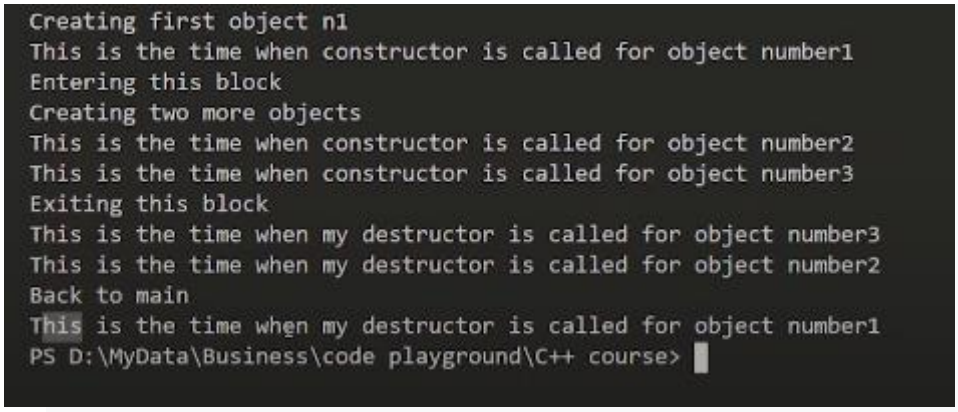


Figure 1: Program Output

As shown in figure 1, first the constructor for the object “n1” was called; second the constructor for the objects “n2” and “n3” was called; third the destructor was called for the objects “n2” and “n3”; at the end destructor for the object “n1” was called.

Inheritance & Its Different Types with Examples in C++ | 36

In this tutorial, we will discuss inheritance in C++

Inheritance in C++ an Overview

- Reusability is a very important feature of OOPs
- In C++ we can reuse a class and add additional features to it
- Reusing classes saves time and money
- Reusing already tested and debugged classes will save a lot of effort of developing and debugging the same thing again

What is Inheritance in C++?

- The concept of reusability in C++ is supported using inheritance
- We can reuse the properties of an existing class by inheriting it
- The existing class is called a base class
- The new class which is inherited from the base class is called a derived class
- Reusing classes saves time and money
- There are different types of inheritance in C++

Forms of Inheritance in C++

- Single Inheritance
- Multiple Inheritance
- Hierarchical Inheritance
- Multilevel Inheritance
- Hybrid Inheritance

Single Inheritance in C++

Single inheritance is a type of inheritance in which a derived class is inherited with only one base class. For example, we have two classes “employee” and “programmer”. If the “programmer” class is inherited from the “employee” class which means that the “programmer” class can now implement the functionalities of the “employee” class.

Multiple Inheritances in C++

Multiple inheritances are a type of inheritance in which one derived class is inherited with more than one base class. For example, we have three classes “employee”, “assistant” and “programmer”. If the “programmer” class is inherited from the “employee” and “assistant” class which means that the “programmer” class can now implement the functionalities of the “employee” and “assistant” class.

Hierarchical Inheritance

A hierarchical inheritance is a type of inheritance in which several derived classes are inherited from a single base class. For example, we have three classes “employee”, “manager” and “programmer”. If the “programmer” and “manager” classes are inherited from the “employee” class which means that the “programmer” and “manager” class can now implement the functionalities of the “employee” class.

Multilevel Inheritance in C++

Multilevel inheritance is a type of inheritance in which one derived class is inherited from another derived class. For example, we have three classes “animal”, “mammal” and “cow”. If the “mammal” class is inherited from the “animal” class and “cow” class is inherited from “mammal” which means that the “mammal” class can now implement the functionalities of “animal” and “cow” class can now implement the functionalities of “mammal” class.

Hybrid Inheritance in C++

Hybrid inheritance is a combination of multiple inheritance and multilevel inheritance. In hybrid inheritance, a class is derived from two classes as in multiple inheritances. However, one of the parent classes is not a base class. For example, we have four classes “animal”, “mammal”, “bird”, and “bat”. If “mammal” and “bird” classes are inherited from the “animal” class and “bat” class is inherited from “mammal” and “bird” classes which means that “mammal” and “bird” classes can now implement the functionalities of “animal” class and “bat” class can now implement the functionalities of “mammal” and “bird” classes.

Inheritance Syntax & Visibility Mode in C++ | 37

In this tutorial, we will discuss inheritance syntax and visibility mode in C++

Inheritance Syntax and Visibility mode in C++

Inheritance is a process of inheriting attributes of the base class by a derived class. The syntax of the derived class is shown below.

```
 // Derived Class syntax
class {{derived-class-name}} : {{visibility-mode}} {{base-class-name}}
{
    class members/methods/etc...
}
```

Copy

Code Snippet 1: Derived Class syntax

As shown in a code snippet 1,

- After writing the class keyword we have to write the derived class name and then put a “:” sign.
- After “:” sign we have to write the visibility mode and then write the base class name.

Note:

- Default visibility mode is private
- Public Visibility Mode: Public members of the base class becomes Public members of the derived class
- Private Visibility Mode: Public members of the base class become private members of the derived class
- Private members are never inherited

An example program is shown below to demonstrate the concept of inheritance.

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

```
// Base Class
class Employee
{
public:
    int id;
    float salary;
    Employee(int inpId)
    {
        id = inpId;
        salary = 34.0;
    }
    Employee() {}
};

// Creating a Programmer class derived from Employee Base class
class Programmer : public Employee
{
public:
    int languageCode;
    Programmer(int inpId)
    {
        id = inpId;
        languageCode = 9;
    }
    void getData(){
        cout<<id<<endl;
    }
};
```

Copy

Code Snippet 2: Inheritance Example Program

As shown in Code snippet 2,

- 1st we created an “employee” class which consists of public data member’s integer “id” and float “salary”.
- 2nd the “employee” class consists of a parameterized constructor that takes an integer “inpid” parameter and assigns its value to the data member “id”. The value of variable “salary” is set to “34”.
- 3rd the “employee” class also consists of default constructor.
- 4th we created a “programmer” class that is inheriting “employee” class. The main thing to note here is that the “visibility-mode” is “public”.
- 5th the “programmer” class consists of public data member’s integer “languageCode”.
- 6th the “programmer” class consists of a parameterized constructor that takes an integer “inpid” parameter and assigns its value to the data member “id”. The value of variable “languageCode” is set to “9”.
- 7th “programmer” class consists of a function “getData” which will print the value of the variable “id”.

The main program is shown in code snippet 3.

```
int main()
{
    Employee harry(1), rohan(2);
    cout << harry.salary << endl;
    cout << rohan.salary << endl;
    Programmer skillF(10);
    cout << skillF.languageCode<<endl;
```

```
    cout << skillF.id<<endl;
    skillF.getData();
    return 0;
}
```

Copy

Code Snippet 3: Main Program

As shown in a code snippet 3,

- 1st objects “harry” and “rohan” is created of the “employee” data type. Object “harry” is passed with the value “1” and the object “rohan” is passed with the value “2”.
- 2nd the “salary” of both objects “rohan” and “harry” are printed.
- 3rd object “skillF” is created of the “programmer” data type. Object “skillF” is passed with the value “10”.
- 4th the “languageCode” and “id” of both object “skillF” is printed.
- 5th the function “getData” is called by the “skillF” object. This will print the “id”.

The output for the following program is shown in figure 1.

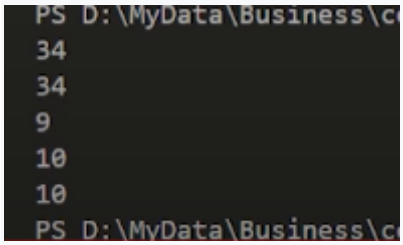


Figure 1: Program Output

Single Inheritance Deep Dive: Examples + Code | 38

In this tutorial, we will discuss single inheritance in C++

Single Inheritance in C++

Single inheritance is a type of inheritance in which a derived class is inherited with only one base class. For example, we have two classes “employee” and “programmer”. If the “programmer” class is inherited from the “employee” class which means that the “programmer” class can now implement the functionalities of the “employee” class.

An example program to demonstrate the concept of single inheritance in C++ is shown below.

```
class Base
{
    int data1; // private by default and is not inheritable
public:
    int data2;
    void setData();
    int getData1();
    int getData2();
};

void Base ::setData(void)
{
    data1 = 10;
```

```
    data2 = 20;
}

int Base::getData1()
{
    return data1;
}

int Base::getData2()
{
    return data2;
}
```

Copy

Code Snippet 1: Base Class

As shown in a code snippet 1,

- 1st we created a “base” class which consists of private data member’s integer “data1” and public data member integer “data2”.
- 2nd the “base” class consists of three member functions “setData”, “getData1”, and “getData2”.
- 3rd the function “setData” will assign the values “10” and “20” to the data members “data1” and “data2”.
- 4th the function “getData1” will return the value of the data member “data1”.
- 5th the function “getData2” will return the value of the data member “data2”.

The derived class will inherit the base class which is shown below.

```
class Derived : public Base
{ // Class is being derived publically
    int data3;

public:
    void process();
    void display();
};

void Derived ::process()
{
    data3 = data2 * getData1();
}

void Derived ::display()
{
    cout << "Value of data 1 is " << getData1() << endl;
    cout << "Value of data 2 is " << data2 << endl;
    cout << "Value of data 3 is " << data3 << endl;
}
```

Copy

Code Snippet 2: Derived Class

As shown in Code snippet 2,

- 1st we created a “derived” class which is inheriting the base class publically. The “derived” class consists of private data member’s integer “data3”.

- 2nd the “derived” class consists of two public member functions “process” and “display”.
- 3rd the function “process” will multiply the values “data2” and “data1”; and store the values in the variable “data3”.
- 4th the function “display” will print the values of the data member “data1”, “data2”, and “data3”.

The main program is shown in code snippet 3.

```
int main()
{
    Derived der;
    der.setData();
    der.process();
    der.display();

    return 0;
}
```

Copy

Code Snippet 3: Main Program

As shown in a code snippet 3,

- 1st object “der” is created of the “derived” data type.
- 2nd the function “setData” is called by the object “der”. This function will set the values of the data members “data1” and “data2”
- 3rd the function “process” is called by the object “der”. This function will multiply the values “data2” and “data1”; and store their value in the variable “data3”.
- 4th the function “display” is called by the object “der”. This function will print the values of the data member “data1”, “data2”, and “data3”.

The output for the following program is shown in figure 1.

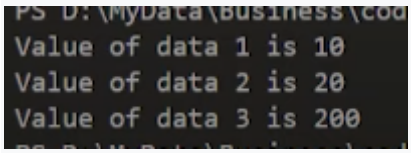


Figure 1: Program Output

Protected Access Modifier in C++ | 39

In this tutorial, we will discuss protected access modifiers in C++

Protected Access Modifiers in C++

Protected access modifiers are similar to the private access modifiers but protected access modifiers can be accessed in the derived class whereas private access modifiers cannot be accessed in the derived class. A table is shown below which shows the behavior of access modifiers when they are derived “public”, “private”, and “protected”.

	ivation	rivation	Derivation
embers	ted	ted	ted
members			
mbers			

As shown in the table,

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

3. If the class is inherited in public mode then its public members are public and can be accessed inside child class and outside the class.
4. If the class is inherited in private mode then its private members cannot be inherited in child class.
5. If the class is inherited in private mode then its protected members are private and cannot be accessed in child class.
6. If the class is inherited in private mode then its public members are private and cannot be accessed in child class.
7. If the class is inherited in protected mode then its private members cannot be inherited in child class.
8. If the class is inherited in protected mode then its protected members are protected and can be accessed in child class.
9. If the class is inherited in protected mode then its public members are protected and can be accessed in child class.

An example program to demonstrate the concept of protected access modifiers is shown below.

```
include<iostream>
using namespace std;


class Base{
    protected:
        int a;
    private:
        int b;


};


class Derived: protected Base{


};


int main(){
    Base b;
    Derived d;
    // cout<<d.a; // Will not work since a is protected in both base as well as derived class
    return 0;
}
```

Copy

Code Snippet 1: Protected Access Modifier Example Program

As shown in a code snippet 1,

- 1st we created a “Base” class which consists of protected data member integer “a” and private data member integer “b”.
- 2nd we created a “Derived” class which is inheriting the “Base” class in protected mode.
- 3rd the object “b” of the data type “Base” is created.
- 4th the object “d” of the data type “Derived” is created.
- 5th if we try to print the value of the data member “a” by using the object “d”; the program will throw an error because the data member “a” is protected and the derived class is inherited in the protected mode. So the data member “a” can only be accessed in the “derived” but not outside the class.

Multilevel Inheritance Deep Dive with Code Example in C++ | 40

In this tutorial, we will discuss multilevel inheritance in C++

Multilevel Inheritance in C++

Multilevel inheritance is a type of inheritance in which one derived class is inherited from another derived class. For example, we have three classes “animal”, “mammal” and “cow”. If the “mammal” class is inherited from the “animal” class and “cow” class is inherited from “mammal” which means that the “mammal” class can now implement the functionalities of “animal” and “cow” class can now implement the functionalities of “mammal” class.

An example program is shown below to demonstrate the concept of multilevel inheritance in C++.

```
include <iostream>
using namespace std;

class Student
{
protected:
    int roll_number;

public:
    void set_roll_number(int);
    void get_roll_number(void);
};

void Student ::set_roll_number(int r)
{
    roll_number = r;
}

void Student ::get_roll_number()
{
    cout << "The roll number is " << roll_number << endl;
}
```

Copy

Code Snippet 1: Student Class

As shown in a code snippet 1,

- 1st we created a “student” class which consists of protected data member integer “roll_number”.
- 2nd the “student” class consists of a public function “set_roll_number” and “get_roll_number”
- 3rd the function “set_roll_number” will set the value of the data member “roll_number”.
- 4th the function “get_roll_number” will print the value of the data member “roll_number”.

The code for the “exam” class is shown below which is inheriting the “student” class

```
class Exam : public Student
{
protected:
    float maths;
    float physics;

public:
    void set_marks(float, float);
    void get_marks(void);
};

void Exam ::set_marks(float m1, float m2)
{
    maths = m1;
    physics = m2;
```

```
}


void Exam ::get_marks()
{
    cout << "The marks obtained in maths are: " << maths << endl;
    cout << "The marks obtained in physics are: " << physics << endl;
}
```

Copy

Code Snippet 2: Exam Class

As shown in Code snippet 2,

- 1st we created an “exam” class that is inheriting “student” class in public mode.
- 2nd the “exam” class consists of protected data members float “math” and float “physics”.
- 3rd the “exam” class consists of public member functions “set_marks” and “get_marks”.
- 4th the function “set_marks” will set the value of the data members “math” and “physics”.
- 5th the function “get_marks” will print the value of the data members “math” and “physics”.

The code for the “result” class is shown below which is inheriting the “exam” class

```
class Result : public Exam
{
    float percentage;

public:
    void display_results()
    {
        get_roll_number();
        get_marks();
        cout << "Your result is " << (maths + physics) / 2 << "%" << endl;
    }
};
```

Copy

Code Snippet 3: Result Class

As shown in a code snippet 3,

- 1st we created a “Result” class which is inheriting the “Exam” class in public mode.
- 2nd the “Result” class consists of private data member’s float “percentage”.
- 3rd the “exam” class consists of the public member function “display_results”.
- 4th the function “display_results” will call the “get_roll_number” and “get_marks” functions, and add the values of “math” and “physics” variables then divide that value with “2” to get a percentage and prints it.

It can be clearly seen that the class “Exam” is inheriting class “student” and class “Results” is inheriting class “Exam”; which is an example of multilevel inheritance. The code main program is shown below.

```
int main()
{
    Result harry;
    harry.set_roll_number(420);
    harry.set_marks(94.0, 90.0);
    harry.display_results();
    return 0;
```

```
}
```

Copy

Code Snippet 4: Main Program

As shown in Code snippet 4,

- 1st object “harry” is created of the “Result” data type.
- 2nd the function “set_roll_number” is called by the object “harry” and the value “420” is passed.
- 3rd the function “set_marks” is called by the object “harry” and the values “94.0” and “90.0” are passed.
- 4th the function “display_results” is called by the object “harry”.

The output for the following program is shown in figure 1.

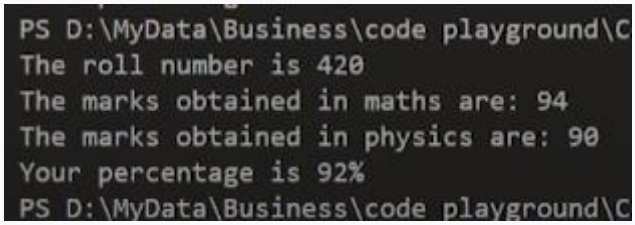


Figure 1: Program Output

Multiple Inheritance Deep Dive with Code Example in C++ | 41

In this tutorial, we will discuss multiple inheritances in C++

Multiple Inheritances in C++

Multiple inheritances are a type of inheritance in which one derived class is inherited with more than one base class. For example, we have three classes “employee”, “assistant” and “programmer”. If the “programmer” class is inherited from the “employee” and “assistant” class which means that the “programmer” class can now implement the functionalities of the “employee” and “assistant” class. The syntax of inheriting multiple inheritances is shown below.

```
// class DerivedC: visibility-mode base1, visibility-mode base2
// {
//      Class body of class "DerivedC"
// };
```

Copy

Code Snippet 1: Multiple inheritances syntax

As shown in a code snippet 1,

- After writing the class keyword we have to write the derived class name and then put a “:” sign.
- After “:” sign we have to write the visibility mode and then write the base class name and again we have to write the visibility mode and write another base class name.

An example program is shown below to demonstrate the concept of multiple inheritances in C++.

```
class Base1{
protected:
    int base1int;


public:
    void set_base1int(int a)
```

```
    {
        base1int = a;
    }
};

class Base2{
protected:
    int base2int;

public:
    void set_base2int(int a)
    {
        base2int = a;
    }
};

class Base3{
protected:
    int base3int;

public:
    void set_base3int(int a)
    {
        base3int = a;
    }
};
```

Copy

Code Snippet 2: Base Classes

As shown in Code snippet 2,

- 1st we created a “Base1” class which consists of protected data member integer “base1int”.
- 2nd the “Base1” class consists of a public function “set_base1int”. This function will set the value of the data member “base1int”.
- 3rd we created a “Base2” class which consists of protected data member integer “base2int”.
- 4th the “Base2” class consists of a public function “set_base2int”. This function will set the value of the data member “base2int”.
- 5th we created a “Base3” class which consists of protected data member integer “base3int”.
- 2nd the “Base3” class consists of a public function “set_base3int”. This function will set the value of the data member “base3int”.

The code for the “Derived” class is shown below. “Derived” class will inherit all the base classes.

```
class Derived : public Base1, public Base2, public Base3
{
    public:
        void show(){
            cout << "The value of Base1 is " << base1int<<endl;
            cout << "The value of Base2 is " << base2int<<endl;
            cout << "The value of Base3 is " << base3int<<endl;
            cout << "The sum of these values is " << base1int + base2int + base3int << endl;
        }
};
```

Copy

Code Snippet 3: Derived Class

As shown in a code snippet 3,

- 1st we created a “Derived” class which is inheriting “Base1”, “Base2”, and “Base3” classes in public mode.
- 2nd the “Derived” class consists of the public member function “show”.
- 4th the function “show” will first print the values of “base1int”, “base2int”, and “base3int” individually and then print the sum of all three values.

It can be clearly seen that the class “Derived” is inheriting class “Base1”, “Base2”, and “Base3”. This is an example of multiple inheritances. The code main program is shown below.

```
int main()
{
    Derived harry;
    harry.set_base1int(25);
    harry.set_base2int(5);
    harry.set_base3int(15);
    harry.show();

    return 0;
}
```

Copy

Code Snippet 4: Main Program

As shown in Code snippet 4,

- 1st object “harry” is created of the “Derived” data type.
- 2nd the function “set_base1int” is called by the object “harry” and the value “25” is passed.
- 3rd the function “set_base2int” is called by the object “harry” and the value “5” is passed.
- 4th the function “set_base3int” is called by the object “harry” and the value “15” is passed.
- 4th the function “show” is called by the object “harry”.

The output for the following program is shown in figure 1.

```
The value of Base1 is 25
The value of Base2 is 5
The value of Base3 is 15
The sum of these values is 45
```

Figure 1: Program Output

Exercise on C++ Inheritance | 42

Exercise on C++ Inheritance

As we have discussed a lot about inheritance and its types; we have also seen its working with example programs. In this tutorial, I will give an exercise on C++ inheritance to be solved.

Questions

You have to create 2 classes:

1. SimpleCalculator - Takes input of 2 numbers using a utility function and performs +, -, *, / and displays the results using another function.

2. ScientificCalculator - Takes input of 2 numbers using a utility function and performs any four scientific operation of your chioice and displays the results using another function.
3. Create another class HybridCalculator and inherit it using these 2 classes

Also, answer the questions given below.

- What type of Inheritance are you using?
- Which mode of Inheritance are you using?
- Create an object of HybridCalculator and display results of simple and scientific calculator.
- How is code reusability implemented?

Code file tut42.cpp as described in the video

```
include<iostream>

using namespace std;

/*

Create 2 classes:

    1. SimpleCalculator - Takes input of 2 numbers using a utility function and performs +, -, *, / and displays the results using another function.

    2. ScientificCalculator - Takes input of 2 numbers using a utility function and performs any four scientific operations of your choice and displays the results using another function.


    Create another class HybridCalculator and inherit it using these 2 classes:

    Q1. What type of Inheritance are you using?

    Q2. Which mode of Inheritance are you using?

    Q3. Create an object of HybridCalculator and display results of the simple and scientific calculator.

    Q4. How is code reusability implemented?

*/

int main(){


    return 0;

}
```

Copy

Ambiguity Resolution in Inheritance in C++ | 43

In this tutorial, we will discuss ambiguity resolution in inheritance in C++

Ambiguity Resolution in Inheritance

Ambiguity in inheritance can be defined as when one class is derived for two or more base classes then there are chances that the base classes have functions with the same name. So it will confuse derived class to choose from similar name functions. To solve this ambiguity scope resolution operator is used "::". An example program is shown below to demonstrate the concept of ambiguity resolution in inheritance.

```
class Base1{

    public:

        void greet(){

            cout<<"How are you?"<<endl;

        }

};


class Base2{

    public:
```

```
        void greet()
        {
            cout << "Kaise ho?" << endl;
        }
};



class Derived : public Base1, public Base2{
   int a;
   public:
    void greet(){
        Base2 :: greet();
    }
};
```

Copy

Code Snippet 1: Ambiguity Resolution in Inheritance Example Program 1

As shown in a code snippet 1,

1. We have created a “Base1” class which consists of public member function “greet”. The function “greet” will print “how are you?”
2. We have created a “Base2” class which consists of public member function “greet”. The function “greet” will print “kaise ho?”
3. We have created a “Derived” class which is inheriting “Base1” and “Base2” classes. The “Derived” class consists of public member function “greet”. The function “greet” will run the “greet” function of the “Base2” class because we have used a scope resolution operator to let the compiler know which function should it run otherwise it will cause ambiguity.

The code of the main function is shown below

```
int main(){
  // Ambibuity 1
     Base1 base1obj;
     Base2 base2obj;
     base1obj.greet();
     base2obj.greet();
     Derived d;
     d.greet();

    return 0;
}
```

Copy

Code Snippet 2: Main program 1

As shown in code snippet 2,

1. Object “base1obj” is created of the “Base1” data type.
2. Object “base3obj” is created of the “Base2” data type.
3. The function “greet” is called by the object “base1obj”.
4. The function “greet” is called by the object “base2obj”.
5. Object “d” is created of the “Derived” data type.
6. The function “greet” is called by the object “d”.

The main thing to note here is that when the function “greet” is called by the object “d” it will run the “greet” function of the “Base2” class because we had specified it using scope resolution operator “::” to get rid ambiguity. The output for the following program is shown in figure 1.

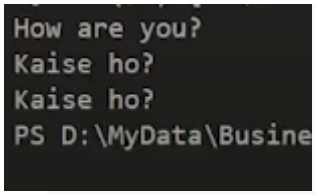


Figure 1: Output

Another example of ambiguity resolution in inheritance is shown below.

```
class B{
    public:
        void say(){
            cout<<"Hello world"<<endl;
        }
};


class D: public B{
    int a;
    // D's new say() method will override base class's say() method
    public:
        void say()
        {
            cout << "Hello my beautiful people" << endl;
        }
};
```

Copy

Code Snippet 3: Ambiguity Resolution in Inheritance Example Program 2

As shown in a code snippet 3,

1. We have created a “B” class which consists of public member function “say”. The function “say” will print “hello world”
2. We have created a “D” class that is inheriting the “B” class. The “D” class consists of the public member function “say”. The function “say” will print “Hello my beautiful people”

The main thing to note here is that both “B” and “D” classes have the same function “say”, So if the class “D” will call the function “say” it will override the base class “say” method because compiler by default run the method which is already written in its own body. But if the function “say” was not present in the class “D” then the compiler will run the method of the class “B”.

The code of the main function is shown below,

```
int main(){
    // Ambibuity 2
    B b;
    b.say();

    D d;
    d.say();

    return 0;
}
```

Copy

Code Snippet 4: Main Program 2

As shown in code snippet 4,

1. Object “b” is created of the “B” data type.
2. The function “say” is called by the object “b”.
3. Object “d” is created of the “D” data type.
4. The function “say” is called by the object “d”.

The output for the following program is shown in figure 2.

```
Hello world
Hello my beautiful people
```

Figure 2: Output

Virtual Base Class in C++ | 44

In this tutorial, we will discuss virtual base class in C++

Virtual Base Class in C++

The virtual base class is a concept used in multiple inheritances to prevent ambiguity between multiple instances. For example: suppose we created a class “A” and two classes “B” and “C”, are being derived from class “A”. But once we create a class “D” which is being derived from class “B” and “C” as shown in figure 1.

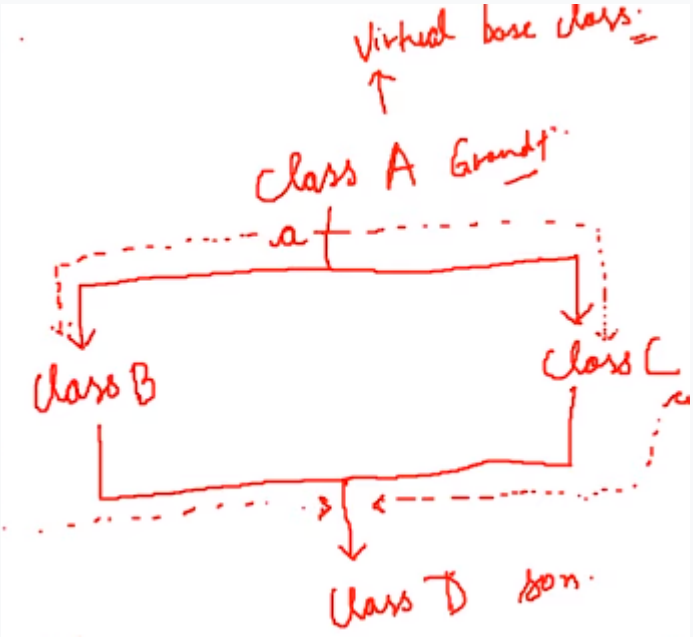


Figure 1: Virtual Base Class Example Diagram

As shown in figure 1,

1. Class “A” is a parent class of two classes “B” and “C”
2. And both “B” and “C” classes are the parent of class “D”

The main thing to note here is that the data members and member functions of class “A” will be inherited twice in class “D” because class “B” and “C” are the parent classes of class “D” and they both are being derived from class “A”.

So when the class “D” will try to access the data member or member function of class “A” it will cause ambiguity for the compiler and the compiler will throw an error. To solve this ambiguity we will make class “A” as a virtual base class. To make a virtual base class “virtual” keyword is used.

When one class is made virtual then only one copy of its data member and member function is passed to the classes inheriting it. So in our example when we will make class “A” a virtual class then only one copy of the data member and member function will be passed to the classes “B” and “C” which will be shared between all classes. This will help to solve the ambiguity.

The syntax of the virtual base class is shown in the code snippet below,

```
include <iostream>
using namespace std;
class A {
public:
    void say()
    {
        cout << "Hello world"<<endl;
    }
};
class B : public virtual A {
};
class C : public virtual A {
};
class D : public B, public C {
};
```

Copy

Code Snippet 1: Virtual Base Class Syntax Example Code

Code Example Demonstrating Virtual Base Class in C++ | 45

In this tutorial, we will discuss demonstrating of virtual base class in C++

Virtual Base Class in C++

The virtual base class is a concept used in multiple inheritances to prevent ambiguity between multiple instances. For example: suppose we created a class “Student” and two classes “Test” and “Sports”, are being derived from class “Student”. But once we create a class “Result” which is being derived from class “Test” and “Sports” as shown in figure 1.

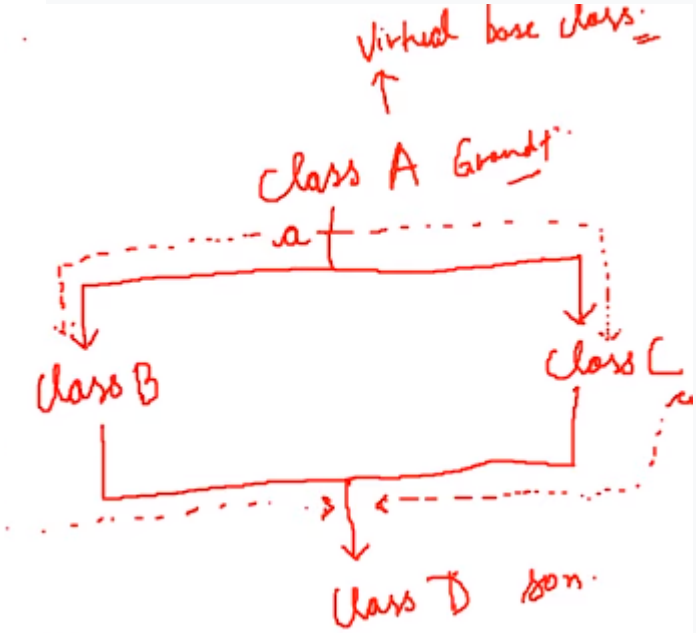


Figure 1: Virtual Base Class Example Diagram

As shown in figure 1,

1. Class “Student” is a parent class of two classes “Test” and “Sports”
2. And both “Test” and “Sports” classes are the parent of class “Result”

The main thing to note here is that the data members and member functions of class “Student” will be inherited twice in class “Result” because class “Test” and “Sports” are the parent classes of class “Result” and they both are being derived from class “Student”.

So when the class “Result” will try to access the data member or member function of class “Student” it will cause ambiguity for the compiler and the compiler will throw an error. To solve this ambiguity we will make class “Student” as a virtual base class. To make a virtual base class “virtual” keyword is used.

When one class is made virtual then only one copy of its data member and member function is passed to the classes inheriting it. So in our example when we will make class “Student” a virtual class then only one copy of data member and member function will be passed to the classes “Test” and “Sports” which will be shared between all classes. This will help to solve the ambiguity.

An example program of the following diagram is shown in a code snippet below,

```
include<iostream>
using namespace std;

class Student{
    protected:
        int roll_no;
    public:
        void set_number(int a){
            roll_no = a;
        }
        void print_number(void){
            cout<<"Your roll no is "<< roll_no<<endl;
        }
};

class Test : public Student{
    protected:
        float maths, physics;
        public:
            void set_marks(float m1, float m2){
                maths = m1;
                physics = m2;
            }

            void print_marks(void){
                cout << "You result is here: "<<endl
                     << "Maths: "<< maths<<endl
                     << "Physics: "<< physics<<endl;
            }
};

class Sports: public Student{
    protected:
        float score;
```

```
    public:
        void set_score(float sc){
            score = sc;
        }

        void print_score(void){
            cout<<"Your PT score is "<<score<<endl;
        }

};

class Result : public Test, public Sports{
    private:
        float total;
    public:
        void display(void){
            total = maths + physics + score;
            print_number();
            print_marks();
            print_score();
            cout<< "Your total score is: "<<total<<endl;
        }
};
```

Copy

Code Snippet 1: Virtual Base Class Example Program

As shown in a code snippet 1,

1. We have created a “Student” class that consists of protected data member “roll_no” and member functions “set_number” and “print_number”. The function “set_number” will assign the value to the protected data member “roll_no” and the function “print_number” will print the value of data member “roll_no”.
2. We have created a “Test” class that is inheriting the virtual base class “Student”. The “Test” consists of protected data members “maths” and “physics” and member functions “set_marks” and “print_marks”. The function “set_number” will assign the values to the protected data members “maths” and “physics” and the function “print_marks” will print the value of data members “maths” and “physics”.
3. We have created a “Sports” class that is inheriting the virtual base class “Student”. The “Sports” consists of protected data member “score” and member functions “set_score” and “print_score”. The function “set_score” will assign the values to the protected data members “score” and “physics” and the function “print_score” will print the value of data members “score”.
4. We have created a “Result” class which is inheriting base classes “Test” and “Sports”. The “Result” consists of protected data member “total” and member functions “display”. The function “display” will first add the values of data members “math”, “physics”, and “score” and assign the value to the protected data members “total” and second the “display” function will call the functions “print_number”, ”print_marks”, and “print_score” and also print the value of the data member “total”.

The code of the main function is shown below,

```
int main(){
    Result harry;
    harry.set_number(4200);
    harry.set_marks(78.9, 99.5);
    harry.set_score(9);
    harry.display();
    return 0;
}
```

Copy

Code Snippet 2: Main Program

As shown in code snippet 2,

1. Object “harry” is created of the “Result” data type.
2. The function “set_number” is called by the object “harry” and the value “4200” is passed.
3. The function “set_marks” is called by the object “harry” and the values “48.9” and “99.5” are passed.
4. The function “set_score” is called by the object “harry” and the value “9” is passed.
5. The function “display” is called by the object “harry”.

The main thing to note here is that there will be no ambiguity because we have made the “Student” class as a virtual base class but if we remove the “virtual” keyword then the compare will throw an error. The output of the following program is shown below.

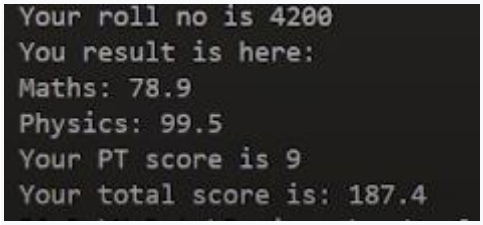


Figure 2: Program Output

Constructors in Derived Class in C++ | 46

In this tutorial, we will discuss constructors in derived class in C++

Constructors in Derived Class in C++

- We can use constructors in derived classes in C++
- If the base class constructor does not have any arguments, there is no need for any constructor in the derived class
- But if there are one or more arguments in the base class constructor, derived class need to pass argument to the base class constructor
- If both base and derived classes have constructors, base class constructor is executed first

Constructors in Multiple Inheritances

- In multiple inheritances, base classes are constructed in the order in which they appear in the class deceleration. For example if there are three classes “A”, “B”, and “C”, and the class “C” is inheriting classes “A” and “B”. If the class “A” is written before class “B” then the constructor of class “A” will be executed first. But if the class “B” is written before class “A” then the constructor of class “B” will be executed first.
- In multilevel inheritance, the constructors are executed in the order of inheritance. For example if there are three classes “A”, “B”, and “C”, and the class “B” is inheriting classes “A” and the class “C” is inheriting classes “B”. Then the constructor will run according to the order of inheritance such as the constructor of class “A” will be called first then the constructor of class “B” will be called and at the end constructor of class “C” will be called.

Special Syntax

- C++ supports a special syntax for passing arguments to multiple base classes
- The constructor of the derived class receives all the arguments at once and then will pass the call to the respective base classes
- The body is called after the constructors is finished executing

Syntax Example:

```
Derived-Constructor (arg1, arg2, arg3….): Base 1-Constructor (arg1,arg2), Base 2-Constructor(arg3,arg4)
{
….
} Base 1-Constructor (arg1,arg2)
```

Copy

Special Case of Virtual Base Class

- The constructors for virtual base classes are invoked before a non-virtual base class

- If there are multiple virtual base classes, they are invoked in the order declared
- Any non-virtual base class are then constructed before the derived class constructor is executed

Solution to Exercise on Cpp Inheritance | 47

A solution to Exercise on Inheritance in C++

As I have given you an exercise on inheritance to solve in the previous tutorial. In this tutorial, we will see the solution to that exercise. So the question was to make three classes “SimpleCalculator”, “ScientificCalculator” and “HybridCalculator”.

- In “SimpleCalculator” class you have to take input of 2 numbers and perform function (+, -, *, /)
- In “ScientificCalculator” class you have to take input of 2 numbers and perform any 4 scientific operations
- You have to inherit both “SimpleCalculator” and “ScientificCalculator” classes with the “HybridCalculator” class. You have to make an object of the “HybridCalculator” class and display the results of “SimpleCalculator” and “ScientificCalculator” classes.

The solution to the above Question is shown below,

```
class SimpleCalculator {
    int a, b;
    public:
        void getDataSimple()
        {
            cout<<"Enter the value of a"<<endl;
            cin>>a;
            cout<<"Enter the value of b"<<endl;
            cin>>b;
        }

        void performOperationsSimple(){
            cout<<"The value of a + b is: "<<a + b<<endl;
            cout<<"The value of a - b is: "<<a - b<<endl;
            cout<<"The value of a * b is: "<<a * b<<endl;
            cout<<"The value of a / b is: "<<a / b<<endl;
        }
};
```

Copy

Code snippet 1: Simple Calculator Class

As shown in a code snippet 1,

1. We created a class “SimpleCalculator” which contains two private data members “a” and “b”
2. The class “SimpleCalculator” contains two member functions “getDataSimple” and “performOperationsSimple”
3. The function “getDataSimple” will take 2 numbers as input
4. The function “performOperationsSimple” will perform the operations “+, -, *, /”

```
class ScientificCalculator{
    int a, b;

    public:
        void getDataScientific()
        {
            cout << "Enter the value of a" << endl;
```

```
            cin >> a;
            cout << "Enter the value of b" << endl;
            cin >> b;
        }

        void performOperationsScientific()
        {
            cout << "The value of cos(a) is: " << cos(a) << endl;
            cout << "The value of sin(a) is: " << sin(a) << endl;
            cout << "The value of exp(a) is: " << exp(a) << endl;
            cout << "The value of tan(a) is: " << tan(a) << endl;
        }
};
```

Copy

Code Snippet 2: Scientific Calculator Class

As shown in code snippet 2,

1. We created a class “ScientificCalculator” which contains two private data members “a” and “b”
2. The class “ScientificCalculator” contains two member functions “getDataScientific” and “performOperationsScientific”
3. The function “getDataScientific” will take 2 numbers as input
4. The function “performOperationsScientific” will perform the operations “cos, sin, exp, tan”

```
class HybridCalculator : public SimpleCalculator, public ScientificCalculator{


};
```

Copy

Code Snippet 3: Hybrid Calculator Class

As shown in code snippet 3, we created a “HybridCalculator” class which is inheriting the “SimpleCalculator” class and “ScientificCalculator” class.

```
int main()
{
    HybridCalculator calc;
    calc.getDataScientific();
    calc.performOperationsScientific();
    calc.getDataSimple();
    calc.performOperationsSimple();

    return 0;
}
```

Copy

Code Snippet 4: Main Program

As shown in code snippet 4,

1. We created an object “calc” of the data type “hybridCalculator”
2. The function “getDataScientific” is called using the object “calc”
3. The function “performOperationsScientific” is called using the object “calc”
4. The function “getDataSimple” is called using the object “calc”
5. The function “performOperationsSimple” is called using the object “calc”

The output of the following program is shown in the figure below,

```
Enter the value of a
3
Enter the value of b
4
The value of cos(a) is: -0.989992
The value of sin(a) is: 0.14112
The value of exp(a) is: 20.0855
The value of tan(a) is: -0.142547
```

Figure 1: Program Output 1

```
Enter the value of a
1
Enter the value of b
4
The value of a + b is: 5
The value of a - b is: -3
The value of a * b is: 4
The value of a / b is: 0
```

Figure 2: Program Output 2

Q1. What type of Inheritance are you using?

Ans. Multiple inheritances

Q2. Which mode of Inheritance are you using?

Ans. public SimpleCalculator, public ScientificCalculator

Code Example: Constructors in Derived Class in Cpp | 48

In this tutorial, we will discuss constructors in derived class with code example in C++

Constructors in Derived Class in C++

As we have discussed before about the constructors in derived class in a code snippet below three cases are given to clarify the execution of constructors.

```
/*
Case1:
class B: public A{
   // Order of execution of constructor -> first A() then B()
};

Case2:
class A: public B, public C{
    // Order of execution of constructor -> B() then C() and A()
};

Case3:
class A: public B, virtual public C{
```

```
    // Order of execution of constructor -> C() then B() and A()
};


*/
```

Copy

Code Snippet 1: Constructors Execution Example Cases

As shown in Code Snippet 1,

1. In case 1, class “B” is inheriting class “A”, so the order of execution will be that first the constructor of class “A” will be executed and then the constructor of class “B” will be executed.
2. In case 2, class “A” is inheriting two classes “B” and “C”, so the order of execution will be that first constructor of class “B” will be executed and then the constructor of class “C” will be executed and at the end constructor of class “A” will be executed.
3. In case 3, class “A” is inheriting two classes “B” and virtual class “C”, so the order of execution will be that first constructor of class “C” will be executed because it is a virtual class and it is given more preference and then the constructor of class “B” will be executed and at the end constructor of class “A” will be executed.

To demonstrate the concept of constructors in derived classes an example program is shown below.

```
class Base1{
    int data1;
    public:
        Base1(int i){
            data1 = i;
            cout<<"Base1 class constructor called"<<endl;
        }
        void printDataBase1(void){
            cout<<"The value of data1 is "<<data1<<endl;
        }
};

class Base2{
    int data2;

    public:
        Base2(int i){
            data2 = i;
            cout << "Base2 class constructor called" << endl;
        }
        void printDataBase2(void){
            cout << "The value of data2 is " << data2 << endl;
        }
};

class Derived: public Base2, public Base1{
    int derived1, derived2;
    public:
        Derived(int a, int b, int c, int d) : Base2(b), Base1(a)
        {
            derived1 = c;
            derived2 = d;
            cout<< "Derived class constructor called"<<endl;
```

```
        }
        void printDataDerived(void)
        {
            cout << "The value of derived1 is " << derived1 << endl;
            cout << "The value of derived2 is " << derived2 << endl;
        }
};
```

Copy

Code Snippet 2: Constructors in Derived Class Example Program

As shown in code snippet 2,

1. We have created a “Base1” class which consists of private data member “data1” and parameterized constructor which takes only one argument and set the value of data member “data1”. The “Base1” class also contains the member function “printDataBase1” which will print the value of data member “data1”.
2. We have created a “Base2” class which consists of private data member “data2” and parameterized constructor which takes only one argument and set the value of data member “data2”. The “Base2” class also contains the member function “printDataBase2” which will print the value of data member “data2”.
3. We have created a “Derived” class that is inheriting base classes “Base1” and “Base2”. The “Derived” class consists of private data members “derived1” and “derived2”. The “Derived” class contains parameterized constructor which calls the “Base1” and “Base2” class constructors to pass the values, it also assigns the values to the data members “derived1” and “derived2”. The “Derived” class also contains member functions “printDataDerived”. The function “printDataDerived” will print the values of the data member “derived1” and “derived2”.

The main thing to note here is that the constructors will be executed in the order in which the classes are being inherited. As in the example program above the “Base2” class is being inherited first and then “Base1” class is being inherited, so the constructor of “Base2” class will be executed first. The main program of the following example code is shown below.

```
int main(){
    Derived harry(1, 2, 3, 4);
    harry.printDataBase1();
    harry.printDataBase2();
    harry.printDataDerived();
    return 0;
}
```

Copy

Code Snippet 3: Main Program

As shown in code snippet 3,

1. Object “harry” is created of the “Derived” data type and the values (1, 2, 3, 4) are passed.
2. The function “printDataBase1” is called by the object “harry”.
3. The function “printDataBase2” is called by the object “harry”.
4. The function “printDataDerived” is called by the object “harry”.

The output of the following program is shown below,

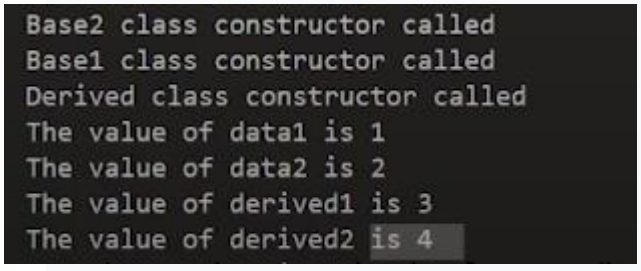


Figure 1: Program Output

Initialization list in Constructors in Cpp | 49

In this tutorial, we will discuss the Initialization list in Constructors in C++

Initialization list in Constructors in C++

The initialization list in constructors is another concept of initializing the data members of the class. The syntax of the initialization list in constructors is shown below.

```
/*
Syntax for initialization list in constructor:
constructor (argument-list) : initilization-section
{
    assignment + other code;
}
```

Copy

Code Snippet 1: Initialization list in Constructors Syntax

As shown in a code snippet 1,

1. A constructor is written first and then the initializations section is written
2. In the initialization section, the data members are initialized

To demonstrate the concept of Initialization list in Constructors an example program is shown below,

```
class Test
{
    int a;
    int b;

public:
    Test(int i, int j) : a(i), b(j)
    {
        cout << "Constructor executed"<<endl;
        cout << "Value of a is "<<a<<endl;
        cout << "Value of b is "<<b<<endl;
    }
};

int main()
{
    Test t(4, 6);

    return 0;
}
```

Copy

Code Snippet 2: Initialization list in Constructors Example Program 1

As shown in code snippet 2,

1. We have created a “test” class that consists of private data member “a” and “b” and parameterized constructor which takes two arguments and sets the value of data member “a” and “b” by using the initialization list. The constructor will also print the value of data member “a” and “b”.
2. In the main program object “t” is created of the “test” data type and the values (4, 6) are passed.

The output of the following program is shown below,

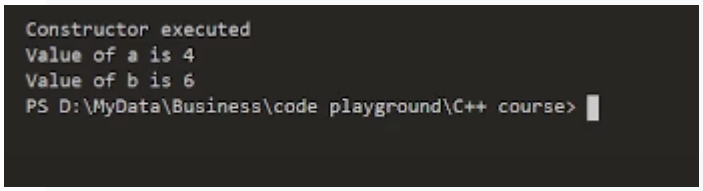


Figure 1: Program Output

Main Points

The main thing to note here is that if we use the code shown below to initialize data members the compiler will throw an error because the data member “a” is being initialized first and the “b” is being initialized second so we have to assign the value to “a” data member first.

```
Test(int i, int j) : b(j), a(i+b)
```

Copy

Code Snippet 3: Initialization list in Constructors Example 1

But if we use the code shown below to initialize data members the compiler will not throw an error because the data member “a” is being initialized first and we are assigning the value to the data member “a” first.

```
Test(int i, int j) : a(i), b(a + j)
```

Copy

Code Snippet 4: Initialization list in Constructors Example 2

Revisiting Pointers: new and delete Keywords in CPP | 50

In this tutorial, we will discuss pointers and new, delete keywords in C++

Pointers in C++

Pointers are variables that are used to store the address. Pointers are created using “*”. An example program of pointers is shown below

```
include<iostream>
using namespace std;

int main(){
    // Basic Example
    int a = 4;
    int* ptr = &a;
    cout<<"The value of a is "<<*(ptr)<<endl;

    return 0;
}
```

Copy

Code Snippet 1: Pointer Example Program 1

As shown in a code snippet 1,

1. We created an integer variable “a” and assign the value “4” to it
2. We created an integer pointer “ptr” and assign the address of variable “a”
3. And printed the value at the address of pointer “ptr”

The output of the following program is shown below,

```
The value of a is 4
PS D:\MyData\Business\c
```

Figure 1: Pointer Program 1 Output

As shown in figure 1, we get the output value “4” because pointer “ptr” is pointing to the variable “a” and the value of the variable “a” is “4” that is why we get the output “4”.

New Keyword

Another example program for pointers and the use of a “new” keyword is shown below.

```
include<iostream>
using namespace std;

int main(){

    float *p = new float(40.78);
    cout << "The value at address p is " << *(p) << endl;

    return 0;
}
```

Copy

Code Snippet 2: Pointer Example Program 2

As shown in code snippet 2,

1. We created a float pointer “p” and dynamically created a float which has value “40.78” and assigned that value to pointer “p”
2. And printed the value at the address of pointer “p”

The output of the following program is shown below,

```
The value at address p is 40.78
```

Figure 2: Pointer Program 2 Output

As shown in figure 2, we get the output value “40.78” because pointer “p” is pointing to an address whose value is “40.78”.

Another example program for pointers array and the use of a “new” keyword with an array is shown below.

```
include<iostream>
using namespace std;

int main(){

    int *arr = new int[3];
    arr[0] = 10;
    arr[1] = 20;
    arr[2] = 30;
    cout << "The value of arr[0] is " << arr[0] << endl;
    cout << "The value of arr[1] is " << arr[1] << endl;
    cout << "The value of arr[2] is " << arr[2] << endl;
```

```
    return 0;
}
```

Copy

Code Snippet 3: Pointer Example Program 3

As shown in a code snippet 3,

1. We created an integer pointer “arr” and dynamically created an array of size three which is assigned to the pointer “arr”
2. The values “10”, “20”, and “30” are assigned to the ”1”, “2”, and “3” indexes of an array
3. And printed the value at the array indexes “1”, “2”, and “3”

The output of the following program is shown below,

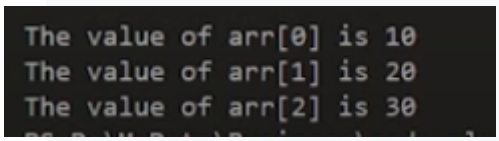


Figure 3: Pointer Program 2 Output

As shown in figure 3, we get the output values “10”, “20”, and “30”.

Delete Keyword

Another example program for pointers array and the use of the “delete” keyword with an array is shown below.

```
include<iostream>
using namespace std;

int main(){

    int *arr = new int[3];
    arr[0] = 10;
    arr[1] = 20;
    arr[2] = 30;
    delete[] arr;
    cout << "The value of arr[0] is " << arr[0] << endl;
    cout << "The value of arr[1] is " << arr[1] << endl;
    cout << "The value of arr[2] is " << arr[2] << endl;

    return 0;
}
```

Copy

Code Snippet 4: Pointer Example Program 4

As shown in code snippet 4,

1. We created an integer pointer “arr” and dynamically created an array of size three which is assigned to the pointer “arr”
2. The values “10”, “20”, and “30” are assigned to the ”1”, “2”, and “3” indexes of an array
3. Before printing the values we used the “delete” keyword
4. And printed the value at the array indexes “1”, “2”, and “3”

The output of the following program is shown below,

```
The value of arr[0] is 15339192
The value of arr[1] is 15335616
The value of arr[2] is 30
```

Figure 4: Pointer Program 2 Output

As shown in figure 2, we get the garbage value in the output instead of “10”, “20”, and “30” because we have used “delete” keyword before printing the values due to which the space used by an array gets free and we get the garbage value in return.

Pointers to Objects and Arrow Operator in CPP | 51

In this tutorial, we will discuss pointers to objects and arrow operator in C++

Pointer to objects in C++

As discussed before pointers are used to store addresses of variables which have data types like int, float, double etc. But pointer can also store the address of an object. An example program is shown below to demonstrate the concept of pointer to objects.

```
include<iostream>
using namespace std;

class Complex{
    int real, imaginary;
    public:
        void getData(){
            cout<<"The real part is "<< real<<endl;
            cout<<"The imaginary part is "<< imaginary<<endl;
        }

        void setData(int a, int b){
            real = a;
            imaginary = b;
        }

};
int main(){
    Complex *ptr = new Complex;
    (*ptr).setData(1, 54); is exactly same as
    (*ptr).getData(); is as good as

    return 0;
}
```

Copy

Code Snippet 1: Pointer to objects Example Program 1

As shown in a code snippet 1,

1. We created a class “Complex”, which contains two private data members “real” and “imaginary”.
2. The class “complex” contains two member functions “getdata” and “setdata”
3. The Function “setdata” will take two parameters and assign the values of parameters to the private data members “real” and “imaginary”
4. The Function “getdata” will print the values of private data members “real” and “imaginary”
5. In the main program object is created dynamically by using the “new” keyword and its address is assigned to the pointer “ptr”
6. The member function “setdata” is called using the pointer “ptr” and the values “1, 54” are passed.

7. The member function “getdata” is called using the pointer “ptr” and it will print the values of data members.

The main thing to note here is that we called the member function with pointers instead of object but still it will give same result because pointer is pointing to the address of that object.

The output of the following program is shown below,

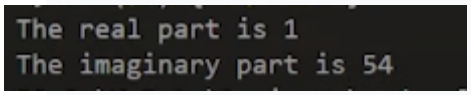


Figure 1: Pointer to Objects Program 1 Output

Arrow Operator in C++

Another example program for the pointer to Objects and the use of the “Arrow” Operator is shown below.

```
include<iostream>
using namespace std;

class Complex{
    int real, imaginary;
    public:
        void getData(){
            cout<<"The real part is "<< real<<endl;
            cout<<"The imaginary part is "<< imaginary<<endl;
        }

        void setData(int a, int b){
            real = a;
            imaginary = b;
        }

};
int main(){
    Complex *ptr = new Complex;
    ptr->setData(1, 54);
    ptr->getData();

    // Array of Objects
    Complex *ptr1 = new Complex[4];
    ptr1->setData(1, 4);
    ptr1->getData();
    return 0;
}
```

Copy

Code Snippet 2: Pointer to Objects with Arrow Operator Example Program 2

As shown in code snippet 2,

1. We created a class “Complex”, which contains two private data members “real” and “imaginary”.
2. The class “complex” contains two member functions “getdata” and “setdata”
3. The Function “setdata” will take two parameters and assign the values of parameters to the private data members “real” and “imaginary”
4. The Function “getdata” will print the values of private data members “real” and “imaginary”

5. In the main program object is created dynamically by using the “new” keyword and its address is assigned to the pointer “ptr”
6. The member function “setdata” is called using the pointer “ptr” with the arrow operator “->” and the values “1, 54” are passed.
7. The member function “getdata” is called using the pointer “ptr” with the arrow operator “->” and it will print the values of data members.
8. Array of objects is created dynamically by using the “new” keyword and its address is assigned to the pointer “ptr1”
9. The member function “setdata” is called using the pointer “ptr1” with the arrow operator “->” and the values “1, 4” are passed.
10. The member function “getdata” is called using the pointer “ptr1” with the arrow operator “->” and it will print the values of data members.

The main thing to note here is that we called the member function with pointers by using arrow operator “->” instead of the dot operator “.” but still it will give the same results.

The output of the following program is shown below,

```
The real part is 1
The imaginary part is 54
The real part is 1
The imaginary part is 4
```

Figure 2: Pointer to Objects Program 2 Output

Array of Objects Using Pointers in C++ | 52

In this tutorial, we will discuss an array of objects using pointers in C++

Array of Objects Using Pointers in C++

Array of objects can be defined as an array that’s each element is an object of the class. In this tutorial, we will use the pointer to store the address of an array of objects. An example program is shown below to demonstrate the concept of an array of objects using pointers.

```
include<iostream>
using namespace std;


class ShopItem
{
    int id;
    float price;
    public:
        void setData(int a, float b){
            id = a;
            price = b;
        }
        void getData(void){
            cout<<"Code of this item is "<< id<<endl;
            cout<<"Price of this item is "<<price<<endl;
        }
};
```

Copy

Code Snippet 1: Array of Objects Using Pointers Example Program

As shown in a code snippet 1,

1. We created a class “ShopItem”, which contains two private data members “id” and “price”.
2. The class “ShopItem” contains two member functions “setdata” and “getdata”
3. The Function “setdata” will take two parameters and assign the values of parameters to the private data members “id” and “price”
4. The Function “getdata” will print the values of private data members “id” and “price”

```
int main(){
    int size = 3;
    ShopItem *ptr = new ShopItem [size];
    ShopItem *ptrTemp = ptr;
    int p, i;
    float q;
    for (i = 0; i < size; i++)
    {
        cout<<"Enter Id and price of item "<< i+1<<endl;
        cin>>p>>q;
        // (*ptr).setData(p, q);
        ptr->setData(p, q);
        ptr++;
    }

    for (i = 0; i < size; i++)
    {
        cout<<"Item number: "<<i+1<<endl;
        ptrTemp->getData();
        ptrTemp++;
    }


    return 0;
}
```

Copy

Code Snippet 2: Main Program

As shown in code snippet 2,

1. We created an integer variable “size” and assigned the value “3” to it.
2. Array of objects of size “3” is created dynamically by using the “new” keyword and its address is assigned to the pointer “ptr”
3. The address of pointer “ptr” is assigned to another pointer “ptrTemp”
4. Two integer variables “p” and “i” are declared and one float variable ”q” is declared
5. We created a “for” loop which will run till the size of array and will take input for “id” and “price” from user at run time. In this “for” loop “setdata” function is called using pointer “ptr”; the function will set the values of “id” and “price” which user will enter. The value of the pointer “ptr” is incremented by 1 in every iteration of loop.
6. We created another “for” loop which will run till the size of array and will print the number of the item. In this “for” loop “getdata” function is called using pointer “ptr”; the function will print the values of “id” and “price”. The value of the pointer “ptrTemp” is incremented by 1 in every iteration of loop.

The main thing to note here is that in the first “for” loop we are incrementing the value of the pointer “ptr” because it is pointing to the address of array of objects and when loop will run every time the function “setdata” will be called by the different object. If we don’t increment the value of the pointer “ptr” the each time function “setdata” will be called by the same object. Likewise in the second loop we are incrementing the pointer “ptrTemp” so that the function “getdata” could be called by each object in the array.

The input and output of the following program is shown below,

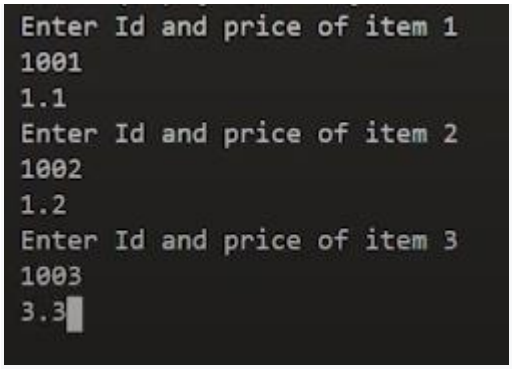


Figure 1: Array of Objects Using Pointer Program Input

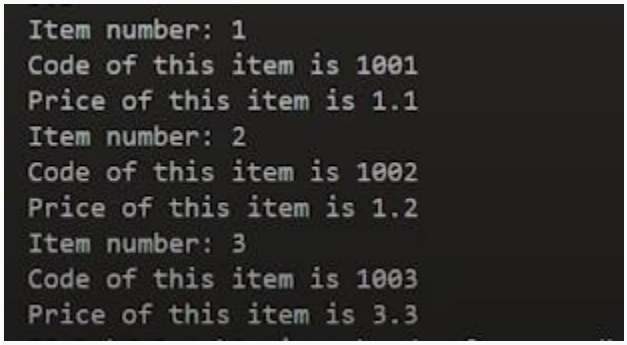


Figure 2: Array of Objects Using Pointer Program Output

this Pointer in C++ | 53

In this tutorial, we will discuss 'this' pointer in C++

'this' Pointer in C++

"this" is a keyword that is an implicit pointer. "this" pointer points to the object which calls the member function. An example program is shown below to demonstrate the concept of "this" pointer.

```
include<iostream>
using namespace std;
class A{
    int a;
    public:
        void setData(int a){
            this->a = a;
        }

        void getData(){
            cout<<"The value of a is "<<a<<endl;
        }
};
```

Copy

Code Snippet 1: "this" Pointer Example Program

As shown in a code snippet 1,

1. We created a class “A”, which contains private data members “a”.
2. The class “A” contains two member functions “setData” and “getData”
3. The Function “setData” will take one parameters and assign the values of parameter to the private data members “a” using “this” pointer. As we know that one copy of member function is shared between all object. The use of “this” pointer helps to points to the object which invokes the member function.
4. The Function “getData” will print the values of private data members “a”

The code for the main program is shown below,

```
int main(){
    A a;
    a.setData(4);
    a.getData();
    return 0;
}
```

Copy

Code Snippet 2: Main Program

As shown in code snippet 2,

1. Object “a” is of data type “A” is created
2. The function “setData” is called using object “a” and the value “4” is passed to the function
3. The function “getData” is called using object “a”

The input and output of the following program is shown below,

```
The value of a is 4
```

Figure 1: Program Output

“this” pointer can be used to return a reference to the invoking object. An example program is shown below.

```
class A{
    int a;
    public:
         A & setData(int a){
            this->a = a;
            return *this;
        }

        void getData(){
            cout<<"The value of a is "<<a<<endl;
        }
};

int main(){
    A a;
    a.setData(4).getData();
    return 0;
}
```

Copy

Code Snippet 3: Return Reference to Invoking Object Example Program

As shown in Code Snippet 3,

1. In the function “setData” the reference of the object is returned using “this” pointer.
2. In the main program by using a single object we have made a chain of the function calls. The main thing to note here is that the function “setData” is returning an object on which we have used the “getData” function. so we don’t need to call the function “getData” explicitly.

Polymorphism in C++ | 54

In this tutorial, we will discuss polymorphism in C++

Polymorphism in C++

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

- Compile-time polymorphism
- Run time polymorphism

Compile Time Polymorphism

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

1. Function Overloading

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

2. Operator Overloading

This is a feature that lets us define operators working for some specific tasks. For example, we can overload the operator “+” and define its functionality to add two strings. Operator loading is also an example of compile-time polymorphism because the compiler already knows at the compile time which operator has to perform the task.

Run Time Polymorphism

In the run-time polymorphism, the compiler doesn’t know already what will happen at run time. Run time polymorphism is also called late binding. The run time polymorphism is considered slow because function calls are decided at run time. Run time polymorphism can be achieved from the virtual function.

3. Virtual Function

A function that is in the parent class but redefined in the child class is called a virtual function. “virtual” keyword is used to declare a virtual function.

Pointers to Derived Classes in C++ | 55

In this tutorial, we will discuss pointer to derived class in C++

Pointer to Derived Class in C++

In C++ we are provided with the functionality to point the pointer to derived class or base class. An example program is shown below to demonstrate the concept of pointer to a derived class in C++

```
include<iostream>
using namespace std;
class BaseClass{
    public:
```

```
        int var_base;
        void display(){
            cout<<"Dispalying Base class variable var_base "<<var_base<<endl;
        }
};


class DerivedClass : public BaseClass{
    public:
            int var_derived;
            void display(){
                cout<<"Dispalying Base class variable var_base "<<var_base<<endl;
                cout<<"Dispalying Derived class variable var_derived "<<var_derived<<endl;
            }
};
```

Copy

Code Snippet 1: Pointer to Derived Class Program Example

As shown in Code snippet 1,

1. We created a class “BaseClass” which contains public data member “var_base” and member function “display”. The member function “display” will print the value of data member “var_base”
2. We created another class “DerivedClass” which is inheriting “BaseClass” and contains data member “var_derived” and member function “display”. The member function “display” will print the values of data members “var_base” and “var_derived”

The code for the main program is shown below,

```
int main(){
    BaseClass * base_class_pointer;
    BaseClass obj_base;
    DerivedClass obj_derived;
    base_class_pointer = &obj_derived; // Pointing base class pointer to derived class


    base_class_pointer->var_base = 34;
    // base_class_pointer->var_derived= 134; // Will throw an error
    base_class_pointer->display();


    base_class_pointer->var_base = 3400;
    base_class_pointer->display();


    DerivedClass * derived_class_pointer;
    derived_class_pointer = &obj_derived;
    derived_class_pointer->var_base = 9448;
    derived_class_pointer->var_derived = 98;
    derived_class_pointer->display();


    return 0;
}
```

Copy

Code Snippet 2: Main Program

As shown in code snippet 2,

1. We created a pointer “base_class_pointer” of the data type “Baseclass”.
2. Object “obj_base” of the data type “BaseClass” is created.
3. Object “obj_derived” of the data type “DerivedClass” is created
4. Pointer “base_class_pointer” of the base class is pointing to the object “obj_derived” of the derived class
5. By using the pointer “base_class_pointer” of the base class we have set the value of the data member “var_base” by “34”. The main thing to note here is that we cannot set the value of the derived class data member by using the base class pointer otherwise the compiler will throw an error.
6. The function “display” is called using a base class pointer. The main thing to note here is that the base class “display” function will run here.
7. Again by using the pointer “base_class_pointer” of the base class we have set the value of the data member “var_base” by “3400” which will update the previous value and the function “display” is called.
8. We created a pointer “derived_class_pointer” of the data type “DerivedClass”
9. Pointer “Derived_class_pointer” of the derived class is pointing to the object “obj_derived” of the derived class
10. By using pointer “Derived_class_pointer” of the derived class we have set the value of the data member “var_base” of the base class by “9448”. The main thing to note here is that this will not throw an error because we can set the value of base class data member by using derived class pointer but we cannot set the value of derived class data member by using base class pointer
11. By using pointer “Derived_class_pointer” of the derived class we have set the value of the data member “var_derived” of the derived class by “98”.
12. The function “display” is called using a derived class pointer. The main thing to note here is that the derived class “display” function will run here.

The output of the following program is shown in figure 1,

```
Dispalying Base class variable var_base 34
Dispalying Base class variable var_base 3400
Dispalying Base class variable var_base 9448
Dispalying Derived class variable var_derived 98
```

Figure 1: Program Output

Virtual Functions in C++ | 56

In this tutorial, we will discuss virtual functions in C++

Virtual Functions in C++

A member function in the base class which is declared using virtual keyword is called virtual functions. They can be redefined in the derived class. To demonstrate the concept of virtual functions an example program is shown below

```
include<iostream>
using namespace std;

class BaseClass{
    public:
        int var_base=1;
        virtual void display(){
            cout<<"1 Dispalying Base class variable var_base "<<var_base<<endl;
        }
};

class DerivedClass : public BaseClass{
    public:
            int var_derived=2;
            void display(){
                cout<<"2 Dispalying Base class variable var_base "<<var_base<<endl;
                cout<<"2 Dispalying Derived class variable var_derived "<<var_derived<<endl;
            }
};
```

Copy

Code Snippet 1: Virtual Function Example Program

As shown in code snippet 1,

1. We created a class “BaseClass” which contains public data member “var_base” which has the value “1” and member function “display”. The member function “display” will print the value of data member “var_base”
2. We created another class “DerivedClass” which is inheriting “BaseClass” and contains data member “var_derived” which has the value “2” and member function “display”. The member function “display” will print the values of data members “var_base” and “var_derived”

The code for the main program is shown below

```
int main(){

    BaseClass * base_class_pointer;

    BaseClass obj_base;

    DerivedClass obj_derived;


    base_class_pointer = &obj_derived;

    base_class_pointer->display();

    return 0;

}
```

Copy

Code Snippet 2: Main Program

As shown in code snippet 2,

1. We created a pointer “base_class_pointer” of the data type “Baseclass”
2. Object “obj_base” of the data type “BaseClass” is created.
3. Object “obj_derived” of the data type “DerivedClass” is created
4. Pointer “base_class_pointer” of the base class is pointing to the object “obj_derived” of the derived class
5. The pointer “base_class_pointer” is pointed to the object “obj_derived” of the derived class.
6. The function “display” is called using the pointer “base_class_pointer” of the base class.

The main thing to note here is that if we don’t use the “virtual” keyword with the “display” function of the base class then beside of the point that we have pointed our base call pointer to derived class object still the compiler would have called the “display” function of the base class because this is its default behavior as we have seen in the previous tutorial.

But we have used the “virtual” keyword with the “display” function of the base class to make is **virtual function** so when the display function is called by using the base class pointer the display function of the derived class will run because the base class pointer is pointing to the derived class object.

The output of the following program is shown in figure 1

```
2 Dispalying Base class variable var_base 1
2 Dispalying Derived class variable var_derived 2
```

Figure 1: Program Output

Virtual Functions Example + Creation Rules in C++ | 57

In this tutorial, we will discuss virtual functions example and its creation rules in C++

Virtual Functions Example in C++

As we have seen in the previous tutorial that how virtual functions are used to implement run-time polymorphism. In this tutorial, we will see an example of virtual functions.

```
class CWH{

    protected:

        string title;
```

```
        float rating;
    public:
        CWH(string s, float r){
            title =  s;
            rating = r;
        }
        virtual void display(){}
};
```

Copy

Code Snippet 1: Code with Harry Class

As shown in a code snippet 1,

1. We created a class “CHW” which contains protected data members “title” which has a “string” data type and “rating” which has a “float” data type.
2. The class “CWH” has a parameterized constructor which takes two parameters “s” and “r” and assign their values to the data members “title” and “rating”
3. The class “CHW” has a virtual function void “display” which does nothing

```
class CWHVideo: public CWH
{
    float videoLength;
    public:
        CWHVideo(string s, float r, float vl): CWH(s, r){
            videoLength = vl;
        }
        void display(){
            cout<<"This is an amazing video with title "<<title<<endl;
            cout<<"Ratings: "<<rating<<" out of 5 stars"<<endl;
            cout<<"Length of this video is: "<<videoLength<<" minutes"<<endl;
        }
};
```

Copy

Code Snippet 2: Code with Harry Video Class

As shown in a code snippet 2,

1. We created a class “CHWVideo” which is inheriting the “CWH” class and contains private data members “videoLength” which has a “float” data type.
2. The class “CWHVideo” has a parameterized constructor which takes three parameters “s”, “r” and “vl”. The constructor of the base class is called in the derived class and the values of the variables “s” and “r” are passed to it. The value of the parameter “vl” will be assigned to the data members “videoLength”
3. The class “CHWVideo” has a function void “display” which will print the values of the data members “title”, “rating” and “videoLength”

```
class CWHText: public CWH
{
    int words;
    public:
        CWHText(string s, float r, int wc): CWH(s, r){
            words = wc;
        }
     void display(){
      cout<<"This is an amazing text tutorial with title "<<title<<endl;
      cout<<"Ratings of this text tutorial: "<<rating<<" out of 5 stars"<<endl;
      cout<<"No of words in this text tutorial is: "<<words<<" words"<<endl;
```

```
        }
};
```

Copy

Code Snippet 3: Code with Harry Text Class

As shown in a code snippet 3,

1. We created a class “CHWText” which is inheriting the “CWH” class and contains private data members “words” which has an “int” data type.
2. The class “CWHText” has a parameterized constructor which takes three parameters “s”, “r” and “wc”. The constructor of the base class is called in the derived class and the values of the variables “s” and “r” are passed to it. The value of the parameter “wc” will be assigned to the data members “words”
3. The class “CHWText” has a function void “display” which will print the values of the data members “title”, “rating” and “words”

```
int main(){
    string title;
    float rating, vlen;
    int words;

    // for Code With Harry Video
    title = "Django tutorial";
    vlen = 4.56;
    rating = 4.89;
    CWHVideo djVideo(title, rating, vlen);

    // for Code With Harry Text
    title = "Django tutorial Text";
    words = 433;
    rating = 4.19;
    CWHText djText(title, rating, words);

    CWH* tuts[2];
    tuts[0] = &djVideo;
    tuts[1] = &djText;

    tuts[0]->display();
    tuts[1]->display();

    return 0;
}
```

Copy

Code Snippet 4: Main Program

As shown in a code snippet 4,

1. We created a string variable “title”, float variables “rating”, “vlen” and integer variable “words”
2. For the code with harry video class, we have assigned “Django tutorial” to the string “title”, “4.56” to the float “vlen” and “4.89” to the float “rating”.
3. An object “djVideo” is created of the data type “CWHVideo” and the variables “title”, “rating” and “vlen” are passed to it.
4. For the code with harry text class, we have assigned “Django tutorial text” to the string “title”, “433” to the integer “words” and “4.19” to the float “rating”.
5. An object “djText” is created of the data type “CWHText” and the variables “title”, “rating” and “words” are passed to it.
6. Two pointers array “tuts” is created of the “CWH” type
7. The address of the “djVideo” is assigned to “tuts[0]” and the address of the “djText” is assigned to “tuts[1]”
8. The function “display” is called using pointers “tuts[0]” and “tuts[1]”

The main thing to note here is that if we don't use the "virtual" keyword with the "display" function of the base class then the "display" function of the base class will run.

But we have used the "virtual" keyword with the "display" function of the base class to make is a **virtual function** so when the display function is called by using the base class pointer the display function of the derived class will run because the base class pointer is pointing to the derived class object.

The output of the following program is shown in figure 1

```
This is an amazing video with title Django tutorial
Ratings: 4.89 out of 5 stars
Length of this video is: 4.56 minutes
This is an amazing text tutorial with title Django tutorial Text
Ratings of this text tutorial: 4.19 out of 5 stars
No of words in this text tutorial is: 433 words
```

Figure 1: Program Output

Rules for virtual functions

1. They cannot be static
2. They are accessed by object pointers
3. Virtual functions can be a friend of another class
4. A virtual function in the base class might not be used.
5. If a virtual function is defined in a base class, there is no necessity of redefining it in the derived class

Abstract Base Class & Pure Virtual Functions in C++ | 58

In this tutorial, we will discuss abstract base class and pure virtual functions in C++

Pure Virtual Functions in C++

Pure virtual function is a function that doesn't perform any operation and the function is declared by assigning the value 0 to it. Pure virtual functions are declared in abstract classes.

Abstract Base Class in C++

Abstract base class is a class that has at least one pure virtual function in its body. The classes which are inheriting the base class must need to override the virtual function of the abstract class otherwise compiler will throw an error.

To demonstrate the concept of abstract class and pure virtual function an example program is shown below.

```
class CWH{
    protected:
        string title;
        float rating;
    public:
        CWH(string s, float r){
            title =  s;
            rating = r;
        }
        virtual void display()=0;
};
```

Copy

Code Snippet 1: Code with Harry Class

As shown in code snippet 1,

1. We created a class “CHW” which contains protected data members “title” which has “string” data type and “rating” which has “float” data type.
2. The class “CWH” has a parameterized constructor which takes two parameters “s” and “r” and assign their values to the data members “title” and “rating”
3. The class “CHW” has a pure virtual function void “display” which is declared by 0. The main thing to note here is that as the “display” function is a pure virtual function it is compulsory to redefine it in the derived classes.

```
class CWHVideo: public CWH

{

    float videoLength;

    public:

        CWHVideo(string s, float r, float vl): CWH(s, r){

            videoLength = vl;

        }

        void display(){

            cout<<"This is an amazing video with title "<<title<<endl;

            cout<<"Ratings: "<<rating<<" out of 5 stars"<<endl;

            cout<<"Length of this video is: "<<videoLength<<" minutes"<<endl;

        }

};
```

Copy

Code Snippet 2: Code with Harry Video Class

As shown in code snippet 2,

1. We created a class “CHWVideo” which is inheriting “CWH” class and contains private data members “videoLength” which has “float” data type.
2. The class “CWHVideo” has a parameterized constructor which takes three parameters “s”, “r” and “vl”. The constructor of the base class is called in the derived class and the values of the variables “s” and “r” are passed to it. The value of the parameter “vl” will be assigned to the data members “videoLength”
3. The class “CHWVideo” has a function void “display” which will print the values of the data members “title”, “rating” and “videoLength”

```
class CWHText: public CWH

{

    int words;

    public:

        CWHText(string s, float r, int wc): CWH(s, r){

            words = wc;

        }

     void display(){

      cout<<"This is an amazing text tutorial with title "<<title<<endl;

      cout<<"Ratings of this text tutorial: "<<rating<<" out of 5 stars"<<endl;

      cout<<"No of words in this text tutorial is: "<<words<<" words"<<endl;

         }

};
```

Copy

Code Snippet 3: Code with Harry Text Class

As shown in code snippet 3,

1. We created a class “CHWText” which is inheriting “CWH” class and contains private data members “words” which has “int” data type.
2. The class “CWHText” has a parameterized constructor which takes three parameters “s”, “r” and “wc”. The constructor of the base class is called in the derived class and the values of the variables “s” and “r” are passed to it. The value of the parameter “wc” will be assigned to the data members “words”
3. The class “CHWText” has a function void “display” which will print the values of the data members “title”, “rating” and “words”

```
int main(){
```

```
    string title;
    float rating, vlen;
    int words;

    // for Code With Harry Video
    title = "Django tutorial";
    vlen = 4.56;
    rating = 4.89;
    CWHVideo djVideo(title, rating, vlen);

    // for Code With Harry Text
    title = "Django tutorial Text";
    words = 433;
    rating = 4.19;
    CWHText djText(title, rating, words);

    CWH* tuts[2];
    tuts[0] = &djVideo;
    tuts[1] = &djText;

    tuts[0]->display();
    tuts[1]->display();

    return 0;
}
```

Copy

Code Snippet 4: Main Program

As shown in code snippet 4,

1. We created a string variable “title”, float variables “rating”, “vlen” and integer variable “words”
2. For the code with harry video class we have assigned “Django tutorial” to the string “title”, “4.56” to the float “vlen” and “4.89” to the float “rating”.
3. An object “djVideo” is created of the data type “CWHVideo” and the variables “title”, “rating” and “vlen” are passed to it.
4. For the code with harry text class we have assigned “Django tutorial text” to the string “title”, “433” to the integer “words” and “4.19” to the float “rating”.
5. An object “djText” is created of the data type “CWHText” and the variables “title”, “rating” and “words” are passed to it.
6. Two pointers array “tuts” is created of the “CWH” type
7. The address of the “djVideo” is assigned to “tuts[0]” and the address of the “djText” is assigned to “tuts[1]”
8. The function “display” is called using pointers “tuts[0]” and “tuts[1]”

The main thing to note here is that if we don’t override the pure virtual function in the derived class the compiler will throw an error as shown in figure 1.

```
tut58.cpp:14:22: note:  'virtual void CWH::display()'
        virtual void display()=0; // do-nothing function --> pure virtual function
                     ^~~~~~~
```

Figure 1: Program Error

The output of the following program is shown in figure 2

```
This is an amazing video with title Django tutorial
Ratings: 4.89 out of 5 stars
Length of this video is: 4.56 minutes
This is an amazing text tutorial with title Django tutorial Text
Ratings of this text tutorial: 4.19 out of 5 stars
No of words in this text tutorial is: 433 words
```

Figure 2: Program Output

File I/O in C++: Working with Files | 59

In this tutorial, we will discuss file input and output in C++

The file is a patent of data which is stored in the disk. Anything written inside the file is called a patent, for example: "**include**" is a patent. The text file is the combination of multiple types of characters, for example, semicolon ";" is a character.

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

The reason that computers can only understand binary numbers is that a computer is made up of switches and switches only perform two operations "true" or "false".

File Input and Output in C++

The file can be of any type whether it is a file of a C++ program, file of a game, or any other type of file. There are two main operations which can be performed on files

- **Read File**
- **Write File**

An image is shown below to show the process of file read and write.

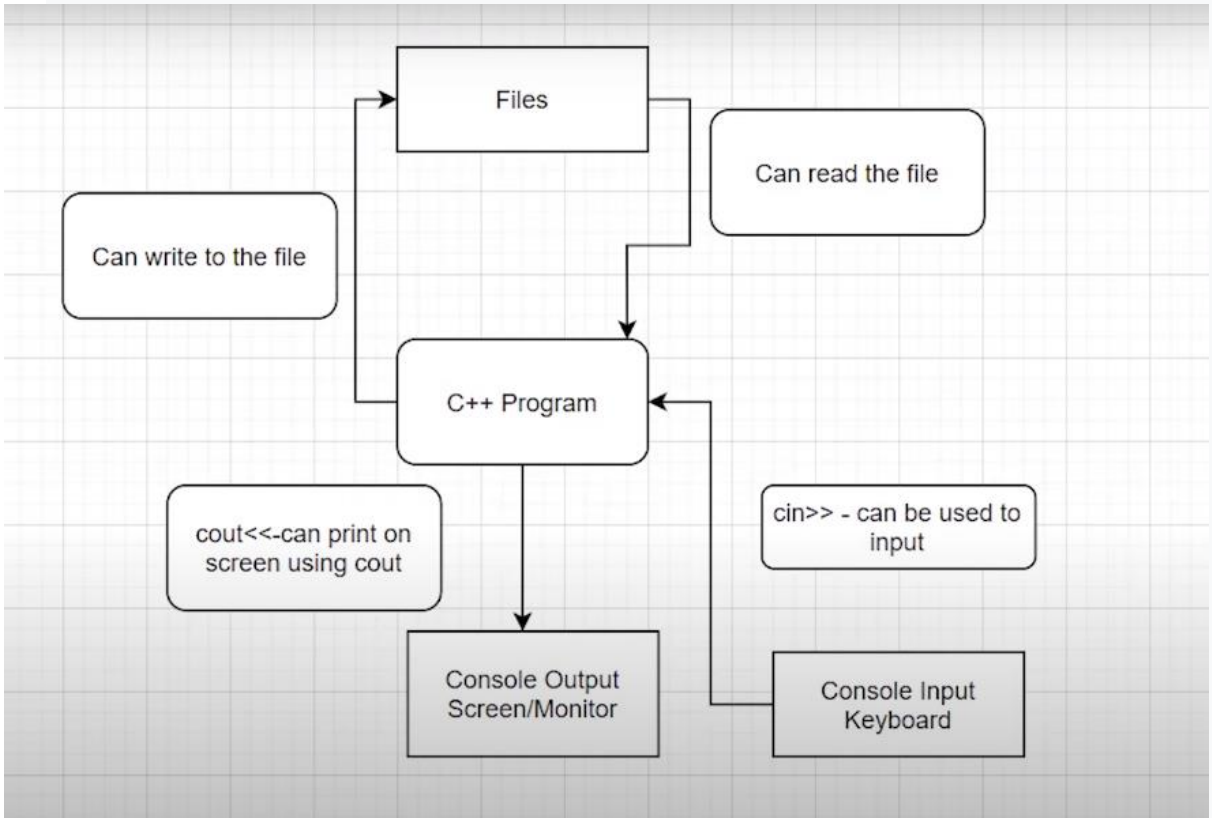


Figure 1: File Read and Write Diagram

As shown in figure 1,

1. The user can provide input to the C++ program by using keyboard through “cin>>” keyword
2. The user can get output from the C++ program on the monitor through “cout<<” keyword
3. The user can write on the file
4. The user can read the file

File I/O in C++: Reading and Writing Files | 60

In this tutorial, we will discuss File I/O in C++: Reading and Writing Files

File I/O in C++: Reading and Writing Files

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

- fstreambase
- ifstream --> derived from fstreambase
- ofstream --> derived from fstreambase

In order to work with files in C++, you will have to open it. Primarily, there are 2 ways to open a file:

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

An example program is shown below to demonstrate the concept of reading and writing files

```
include<iostream>
include<fstream>

using namespace std;

int main(){
    string st = "Harry bhai";
    // Opening files using constructor and writing it
    ofstream out("sample60.txt"); // Write operation
    out<<st;

    return 0;
}
```

Copy

Code Snippet 1: Writing Files Example Program

As shown in a code snippet 1,

1. We have created a string “st” which has a value “harry Bhai”
2. Object “out” is created of the type ofstream and the file “sample60.txt” is passed to it
3. The string “st” is passed to object “out”

The output of the following program is shown in figure 1

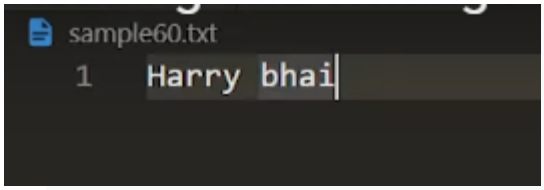


Figure 1: Writing File Operation Output

```
include<iostream>
include<fstream>


using namespace std;


int main(){
    string st2;
    // Opening files using constructor and reading it
    ifstream in("sample60b.txt"); // Read operation
    in>>st2;
    getline(in, st2);
    cout<<st2;


    return 0;
}
```

Copy

Code Snippet 2: Reading Files Example Program

As shown in a code snippet 1,

1. We have created a string “st2” which is empty
2. We have made a text file “sample60b.txt” and written “This is coming from a file” in it
3. Object “in” is created of the type instream and the file “sample60b.txt” is passed to it
4. The function “getline” is called and the object “in” and the string “st2” are passed to it. The main thing to note here is that the function “getline” is used when we want to read the whole line
5. String “st2” is printed

The output of the following program is shown in figure 2

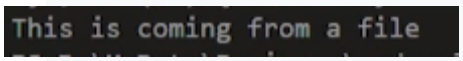


Figure 2: Reading File Operation Output

File I/O in C++: Read/Write in the Same Program & Closing Files | 61

In this tutorial, we’ll learn about creating a program that will read from a file and write to the file in the same program using a constructor.

Before jumping on to the main thing, we’ll first give ourselves a quick revision of the things we had learned previously.

We had learned about the three most useful classes when we talk about File I/O, namely,

1. fstreambase
2. ifstream
3. ofstream.

All the above three classes can be used in a program by first including the header file, fstream.

Reading File Operation Output*:*

We learnt reading from a file using ifstream. Below snippet will help you recollect the same.

```
    string st;
  // Opening files using constructor and reading it
    ifstream in("this.txt"); // Read operation
    in>>st;
```

Copy

Writing File Operation Output*:*

We learnt reading from a file using ofstream. Below snippet will help you recollect the same.

```
    string st = "Harry bhai";
    // Opening files using constructor and writing it
    ofstream out("this.txt"); // Write operation
    out<<st;
```

Copy

Let me make these codes functional in the same program for you to easily understand the workflow.

Suppose we have a file named sample60.txt in the same directory, we can easily call the file infinite number of times in the same program only by maintaining different connections for different purposes, using

```
<object_name>.close();
```

Copy

Now, let's move on to our systems. Open your editors as well. Don't forget to include the header file, <fstream>.

Follow these steps below to first write into the empty file:

1. Create a text file "sample60.txt" in the same directory as that of the program.
2. Create a string variable *name*.
3. Create an object *hout*(name it whatever you wish) using ofstream passing the text file, sample60.txt into it. This establishes a connection between your program and the text file.
4. Take input from the user using cin into the name string.(You can write manually as well)
5. Pass this name string to the object *hout.* The string name gets written in the text file.
6. Disconnect the file with the program since we are done writing to it using *hout.close().*

Since the file has been disconnected from the program, we can connect it again for any other purpose in the same program independently.

Follow these steps below to read from the file we just wrote into:

1. Create a string variable *content*.
2. Create an object *hin*(name it whatever you wish) using ifstream passing the text file, sample60.txt into it. This establishes a new connection between your program and the text file.
3. Fill in the string using the object *hin.* (Use getline, which we talked about in the last video, to take into input the whole line from the text file.)
4. Give output to the user, the string we filled in with the content in the text file.
5. Disconnect the file with the program since we are done reading from it using *hin.close().*

```
include<iostream>
include<fstream>

using namespace std;

int main(){

              // connecting our file with hout stream
               ofstream hout("sample60.txt");
```

```
    // creating a name string variable and filling it with string entered by the user
    string name;
     cout<<"Enter your name: ";
    cin>>name;

   // writing a string to the file
   hout<<name + " is my name";

  // disconnecting our file
  hout.close();
  // connecting our file with hin stream
 ifstream hin("sample60.txt");

// creating a content string variable and filling it with string present there in the text file
 string content;
  hin>>content;
 cout<<"The content of the file is: "<<content;

 // disconnecting our file
 hin.close();
 return 0;
}
```

Copy

Let's run the program we just created, The output will look like this:

```
Enter your name: Harry
The content of the file is: Harry
```

Copy

So when we input a string "Harry" into the text file, it gets written there in the file as below, and when we read it from the file, it gives output as below. Since we used hin and not getline, it could read just the first word.

```
The content of the file is: Harry
```

Copy

Thank you, friends for being with me throughout, hope you liked the tutorial. If you haven't checked out the whole playlist yet, move on to codewithharry.com or my YouTube channel to access it. I hope you enjoy them.

In the next tutorial, we'll be covering the use of open() and eof() functions, see you there, till then keep coding.

File I/O in C++: open() and eof() functions | 62

In this tutorial, we are going to learn about the member functions open and eof of the objects we learnt about previously.

I remember teaching you all about the two methods to open a text file in our C++ program, first one using a constructor which we discussed in the last tutorial, and the second one, using the member function open, which is to be dealt with today.

Using the member function open:

The member function open is used to connect the text file to the C++ program when passed into it.

Understanding the snippet below:

1. Unlike what we did earlier passing the text file in the object while creating it, we'll first just declare an object out(any name you wish) of the type ofstream and use its open method to open the text file in the program.
2. We'll pass some string lines to the text file using the out operation.
3. We'll now close the file using the close function. Now closing is explicitly used to make the system know that we are done with the file. It is always good to use this.

This was all about writing to a file. We'll now move to the eof function's vitality in File I/O.

```
include <iostream>
include <fstream>


using namespace std;


int main()
{


    // declaring an object of the type ofstream
    ofstream out;


    //connecting the object out to the text file using the member function open()
    out.open("sample60.txt");


    //writing to the file
    out <<"This is me\n";
    out <<"This is also me";
    //closing the file connection
    out.close();
    return 0;
}
```

Copy

Using the member function eof:

The member function eof(End-of-file) returns a boolean true if the file reaches the end of it and false if not.

Understanding the snippet below:

1. We'll first declare an object in(any name you wish) of the type ifstream and use its open method similar to what we did above, to open the text file in the program.
2. And now, we'll declare the string variable st to store the content we'll receive from the text file sample60.txt.
3. Now since we not only want the first or some two or three strings present in the text file, but the whole of it, and we have no idea of what the length of the file is, we'll use a while loop.
4. We'll run the while loop until the file reaches the end of it, and that gets checked by using eof() , which returns 1 or true if the file reaches the end. Till then a 0 or false.
5. We'll use getline to store the whole line in the string variable st. Don't forget to include the header file <string>.
6. This program now successfully prints the whole content of the text file.

Refer to the output below the snippet.

```
include <iostream>
include <fstream>
include <string>
```

```cpp
using namespace std;


int main()
{
    // declaring an object of the type ifstream
    ifstream in;
    //declaring string variable st
    string st;
    //opening the text file into in
    in.open("sample60.txt");


    // giving output the string lines by storing in st until the file reaches the end of it
    while (in.eof()==0)
    {
        // using getline to fill the whole line in st
        getline(in,st);
        cout<<st<<endl;
    }
    return 0;
}
```

Copy

Output of the above program:

```
This is me
This is also me
```

Copy

So, this was all about File I/O in C++. Learning to detect the eof and opening files in a C++ program in two ways, writing to it and reading from the same, was all a big deal, and you successfully completed them all. Cheers.

Thank you, friends, for being with me throughout, hope you liked the tutorial. If you haven't checked out the whole playlist yet, move on to codewithharry.com or my YouTube channel to access it. I hope you enjoy them.

In the next tutorial, we'll be starting a topic, a must for competitive programmers, C++ templates, see you there, till then keep coding.

C++ Templates: Must for Competitive Programming | 63

It has been quite a journey till here, and I feel grateful to have you all with me in the same. We have covered a lot in C++ and there is yet a great deal left. But we'll make everything ahead a cakewalk together.

Today we have in the box, the most important topic for all you enthusiastic programmers, C++ templates. We'll follow the below-mentioned roadmap:

1. What is a template in C++ programming?
2. Why templates?
3. Syntax

What is a template in C++ programming?

A template is believed to escalate the potential of C++ several fold by giving it the ability to define data types as parameters making it useful to reduce repetitions of the same declaration of classes for different data types. Declaring classes for every other data type(which if counted is way too much) in the very first place violates the DRY(Don't Repeat Yourself) rule of programming and on the other doesn't completely utilise the potential of C++.

It is very analogous to when we said classes are the templates for objects, here templates itself are the templates of the classes. That is, what classes are for objects, templates are for classes.

Why templates?

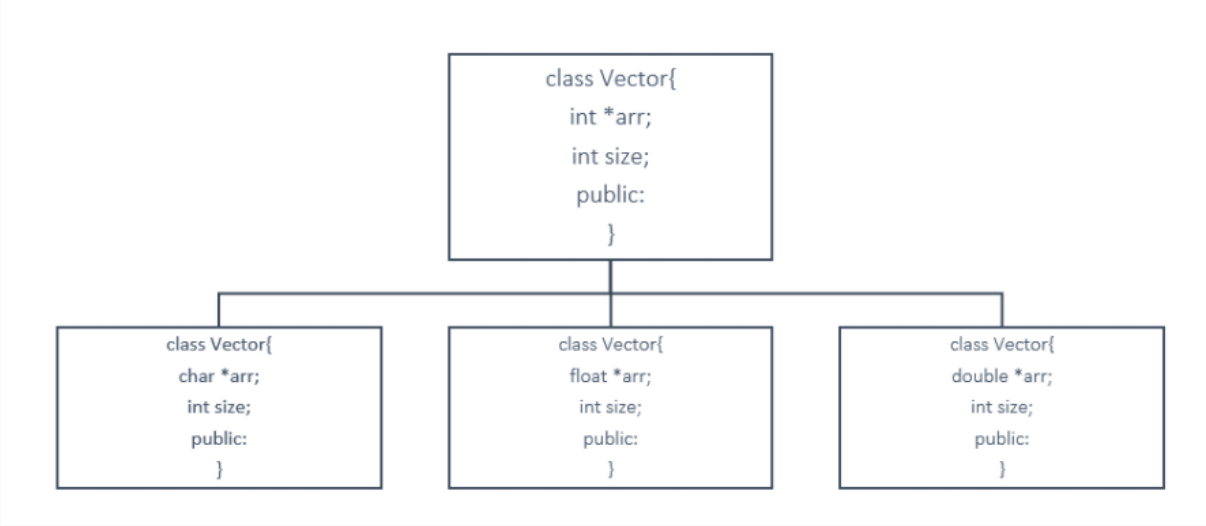
1. **DRY Rule:**

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

1. **Generic Programming:**

It is called generic, because it is sufficient to declare a template once, it becomes general and it works all along for all the data types.

Refer to the schematic below:



We had to copy the same thing again and again for different data types, but a template solves it all. Refer to the syntax section for how.

Below is the template for a vector of int data type, and it goes similarly for float char double, etc.

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

Copy

Syntax:

Understanding the syntax below:

1. First, we declare a template of class and pass a variable T as its parameter.
2. Define the class of vector and keep the data type of *arr as T only. Now, the array becomes of the type we supply in the template.

Now we can easily use this template to declare umpteen number of classes in our main scope. Be it int, float, or arr vector.

```
include <iostream>
using namespace std;

template <class T>
```

```
class vector {
    T *arr;
    int size;
    public:
        vector(T* arr)[
            //code
        ]
        //and many other methods

};

int main() {
    vector<int> myVec1();
    vector<float> myVec2();
    return 0;
}
```

Copy

Templates are believed to be very useful for people who pursue competitive programming. It makes their work several folds easier. It gives them an edge over others. It is a must because it saves you a lot of time while programming. And I believe you ain't want to miss this opportunity to learn, right?

So, get to the playlist as soon as you can. Save yourselves some time and get over your competitors.

Thank you, friends, for being with me throughout, hope you liked the tutorial. And If you haven't checked out the whole playlist yet, it's never too late, move on to codewithharry.com or my YouTube channel to access it. I hope you enjoy them. Templates are an inevitable part of this process of learning C++. You just cannot afford to miss this. In the next tutorial, we'll be writing a program using templates for your better understanding, see you there, till then keep coding.

Writing our First C++ Template in VS Code | 64

In the last tutorial, we learnt about what a template is, why a template is used in programming and what its syntax is. Let's give ourselves a quick revision of everything about templates.

Long story short, a template does the same thing to a class, what a class does to the objects. It parametrizes the data type hence making it easy for us to use different classes without having to write the whole thing again and again, violating the DRY rule. Templates furthermore give our program a generic view, where declaring one template suffices the task.

Today, we'll learn to make a program using templates to give you a better understanding about its uses. I'll make the process effortless for you to learn, so, you stay calm and keep learning.

Now suppose we have two integer vectors and we want to calculate their Dot Product. This part should not be troublesome since we have learnt pretty well the use of classes and constructors. We had learnt to write the code like the one mentioned below.

Understanding the code below to calculate the DotProduct of two integer vectors:

1. Here we declare a class vector, with an integer pointer arr.
2. We declared an integer variable to store the size.
3. We made the constructor for the integer vector. These things should be unchallenging for you by now as they have been already taught.
4. We then wrote a function which returns an integer value, to calculate the Dot Product and named it dotProduct which will take a vector as a parameter.
5. We traversed through the vectors multiplying their corresponding elements and adding it to the sum variable named d.
6. We finally returned it to the main.
7. And the output we received is this:

```
5

PS D:\MyData\Business\code playground\C++ course>
```

Copy

```
include <iostream>

using namespace std;
```

```
class vector
{
    public:
        int *arr;
        int size;
        vector(int m)
        {
            size = m;
            arr = new int[size];
        }
    int dotProduct(vector &v){
        int d=0;
        for (int i = 0; i < size; i++)
        {
            d+=this->arr[i]*v.arr[i];
        }
        return d;
    }
};

int main()
{
    vector v1(3); //vector 1
    v1.arr[0] = 4;
    v1.arr[1] = 3;
    v1.arr[2] = 1;
    vector v2(3); //vector 2
    v2.arr[0]=1;
    v2.arr[1]=0;
    v2.arr[2]=1;
    int a = v1.dotProduct(v2);
    cout<<a<<endl;
    return 0;
}
```

Copy

So, this was all about creating a class and an embedded function to calculate the dot product of two integer vectors. But this program would obviously fail to calculate the dot products for some different data types. It would demand an entirely different class. But we'll save ourselves the effort and the time by declaring a template. Let's see how:

Understanding the changes, we made in the above program to generalise it for all data types:

1. First and foremost, we defined a template with class T where T acts as a variable data type.
2. We then changed the data type of arr to T, changed its constructor to T from int, changed everything except the size of the vector, to a variable T. The function then returned T. This has now changed the class from specific to general.
3. We then very easily added a parameter, while defining the vectors, of its data type. And the compiler itself transformed the class accordingly. Here we passed a float and the code handled it very efficiently.
4. The output we received was:

```
6.82
PS D:\MyData\Business\code playground\C++ course>
```

Copy

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

template <class T>
class vector
{
    public:
        T *arr;
        int size;
        vector(int m)
        {
            size = m;
            arr = new T[size];
        }
    T dotProduct(vector &v){
        T d=0;
        for (int i = 0; i < size; i++)
        {
            d+=this->arr[i]*v.arr[i];
        }
        return d;
    }
};

int main()
{
    vector<float> v1(3); //vector 1 with a float data type
    v1.arr[0] = 1.4;
    v1.arr[1] = 3.3;
    v1.arr[2] = 0.1;
    vector<float> v2(3); //vector 2 with a float data type
    v2.arr[0]=0.1;
    v2.arr[1]=1.90;
    v2.arr[2]=4.1;
    float a = v1.dotProduct(v2);
    cout<<a<<endl;
    return 0;
}
```

Copy

Imagine how tough it would have been without these templates, you'd have made different classes for different data types handling them clumsily increasing your efforts and proportionally your chances of making errors. So, this is a life saviour.

And learning it will only benefit you. So why not.

Thank you, friends for being with me throughout, hope you liked the tutorial. If you haven't checked out the whole playlist yet, move on to codewithharry.com or my YouTube channel to access it. I hope you enjoy them. In the next tutorial, we'll be learning about further uses of a template and multiple parameters, see you there, till then keep coding.

C++ Templates: Templates with Multiple Parameters | 65

In the last tutorial, we had ample understanding of a template and its uses. We had created a template which would calculate the Dot Product of two vectors of any data type just by declaring a simple template parameterizing the data type we usually hardcoded in the classes. This already made our task easier but here we are, with our next tutorial focusing on how to handle multiple parameters in a template.

To give you a short overview of how templates work with multiple parameters, you can think of it as a function where you have that power to pass different parameters of the same or different data types. A simple template with two parameters would look something like this. The only effort it demands is the declaration of parameters. We'll get through it thoroughly by making a real program, so, let's go.

```
include<iostream>
using namespace std;


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


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

Copy

Code Snippet 1: Syntax of a template with multiple parameter

Suppose we have a class named myClass which has two data in it of data types int and char respectively, and the function embedded just displays the two. Fair enough, no big deal, we'll construct our class something like this. The problem arises when we wish to have both our data types anonymous and to be put from the main itself. You will be surprised to know that very subtle modifications in yesterday's code would do our task. Instead of declaring a single parameter T, we would declare two of them namely T1 And T2.

```
class myClass{
    public:
        int data1;
        char data2;
    void display(){
        cout<<this->data1<<" "<<this->data2;
    }
};
```

Copy

Code Snippet 2: Constructing a class

Refer to changes we have done below to parametrize both our data types using a single template:

1. We have declared data1 and data2 with data types T1 and T2 respectively.
2. We have applied the constructor filling the values we receive from the main into data1 and data2.
3. Finally, we have displayed both of them.

```
template<class T1, class T2>
class myClass{
    public:
        T1 data1;
        T2 data2;
        myClass(T1 a,T2 b){
            data1 = a;
```

```
            data2 = b;
        }
    void display(){
        cout<<this->data1<<" "<<this->data2;
    }
};
```

Copy

Code Snippet 2: Constructing a template with two parameters.

Let me now show you how this template works for different parameters. I'll pass different data types from the main and see if it's flexible enough.

Firstly, we put an integer and a char,

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

Copy

Code Snippet 3: Specifying the data types to be int and char.

And the output received was this, which is correct. Let's feed another one.

```
1   c
PS D:\MyData\Business\code playground\C++ course>
```

Copy

Figure 1: Output of code snippet 3.

Now we put an integer and a float,

```
int main()
{
    myClass<int, float> obj(1,1.8 );
    obj.display();
}
```

Copy

Code Snippet 4: Specifying the data types to be int and float.

And the output received was this,

```
1   1.8
PS D:\MyData\Business\code playground\C++ course>
```

Copy

Figure 1: Output of code snippet 4.

So yes, this is functioning all good.

And this was all about templates with multiple parameters, just don't miss out the commas while defining the parameters in a template. And you can have 2, 3 or more of them according to your needs. Could you believe how luxurious it has become to work with customized data types? It is now you, who'll decide what the data type of some variable in a class should be. It is no longer pre-specified. It has given you some unimaginable power which, if you realise, can save you a lot of energy and time.

Thank you, friends for being with me throughout, hope you liked the tutorial. If you haven't checked out the whole playlist yet, move on to codewithharry.com or my YouTube channel to access it. I hope you enjoy them. In the next tutorial, we'll be learning about having a default parameter in a template, see you there, till then keep coding.

C++ Templates: Class Templates with Default Parameters | 66

So far, we have already covered the C++ templates with single parameters. In the last tutorial, we learnt about templates with multiple parameters, when it comes to handling different data types of two or more containers.

Today, we'll be learning a very easy yet powerful attribute of templates, its ability to have default parameters. Its ability to have default specifications about the data type, when it receives no arguments from the main.

So, let's start by making a program manifesting the use of default parameters in a C++ template. **Refer to the code snippet below and follow the steps:**

1. We'll start by constructing a class named Harry.
2. We'll then define a template with any number of arguments, let three, T1, T2, and T3. If you remember, we had this feature of specifying default arguments for functions, similarly we'll mention the default parameters, let, int, float and char for T1, T2 and T3 respectively.
3. This ensures that if the user doesn't put any data type in main, default ones get considered.
4. In public, we'll define variables a, b and c of the variable data types T1, T2 and T3. And build their constructors.
5. The constructor accepts the values featured by the main, and assigns them to our class variables a, b and c. If the user specifies the data types along with the values, the compiler assigns them to T1 , T2 and T3, otherwise gives them the default ones, as specified while declaring the template itself.
6. We'll then create a void function display, just to print the values the user inputs.

```
include<iostream>
using namespace std;

template <class T1=int, class T2=float, class T3=char>
class Harry{
    public:
        T1 a;
        T2 b;
        T3 c;
        Harry(T1 x, T2 y, T3 z) {
            a = x;
            b = y;
            c = z;
        }
        void display(){
            cout<<"The value of a is "<<a<<endl;
            cout<<"The value of b is "<<b<<endl;
            cout<<"The value of c is "<<c<<endl;
        }
};
```

Copy

Since we are done defining the templates and class, we can very easily move to the main where we'll see how these work. **Understanding code snippet 2:**

1. Firstly, we'll create an object, let's name it h, of the class Harry. And we'll pass into it three values, an int, a float and a char, suppose 4, 6.4 and c respectively. Now since we have not specified the data types of the values we have just entered, the default data types, int, float and char would be considered.
2. We'll then display the values, which you'll be seeing when we run the same.
3. And then we'll create another object g, of the class Harry but this time, with the data types of our choice. Let's specify them to be float, char and char.

4. We can then pass some values into it, suppose 1.6, o, and c and call the display function again.
5. These objects are sufficient to give us the main concept behind using a default parameter and the variety of classes we could make via this one template.

```
int main()
{
    Harry<> h(4, 6.4, 'c');
    h.display();
    cout << endl;
    Harry<float, char, char> g(1.6, 'o', 'c');
    g.display();
    return 0;
}
```

Copy

We'll now refer to the output the above codes combinedly gave. As you can see below, it worked all fine. Had we not specified the default parameters; the above program would have thrown an error. Thanks to this feature of C++ templates.

```
The value of a is 4
The value of b is 6.4
The value of c is c

The value of a is 1.6
The value of b is o
The value of c is c
PS D:\MyData\Business\code playground\C++ course>
```

Copy

Thank you, friends for being with me throughout, hope you liked the tutorial. If you haven't checked out the whole playlist yet, move on to codewithharry.com or my YouTube channel to access it. I hope you enjoy them all. So far, we were learning about the class templates. See you all in the next tutorial where we'll look after the **function templates**. Till then keep coding.

C++ Function Templates & Function Templates with Parameters | 67

In this tutorial, we are wishing to learn how a function template works. Prior to this video, we have only talked about a class template and its functionalities. In class template we used to have template parameters which we, very often, addressed as a variable for our data types. We have also declared a class template similar to what shown here below :

```
template <class T1 = int, class T2 = float>
```

Copy

Today, we'll be interested in knowing what a function template does. So. let's get ourselves on our editors.

Suppose we want to have a function which calculates the average of two integers. So, this must be very easy for you to formulate. Look for the snippet below.

1. We have declared a float function named funcAverage which will have two integers as its parameters, a and b.
2. We stored its average in a float variable avg and returned the same to the main.
3. Later we called this function by value, and stored the returned float in a float variable a and printed the same.
4. So this was the small effort we had to make to get a function which calculates the average of two integers.

```
include<iostream>
using namespace std;

float funcAverage(int a, int b){
    float avg= (a+b)/2.0;
    return avg;
}
```

```
int main(){
    float a;
    a = funcAverage(5,2);
    printf("The average of these numbers is %f",a);
    return 0;
}
```

Copy

The output of the above program is :

```
The average of these numbers is 3.500000
PS D:\MyData\Business\code playground\C++ course>
```

Copy

But the effort we made here defining a single function for two integers increases several folds when we demand for a similar function for two floats, or one float and one integer or many more data type combinations. We just cannot repeat the procedure and violate our DRY rule. We'll use function templates very similar to what we did when we had to avoid defining more classes.

See what are the subtle changes we had to make, to make this function generic.

We'll first declare a template with two data type parameters T1 and T2. And replace the data types we mentioned in the function with them. And that's it. Our function has become general for all sorts of data types. Refer to the snippet below.

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

Copy

Let's call this function by passing into it two sorts of data types combination, first, two integers and then one integer and one float. And see if the outputs are correct.

```
int main(){
    float a;
    a = funcAverage(5,2);
    printf("The average of these numbers is %f",a);
    return 0;
}
```

Copy

Code snippet: Calling the function by passing two integers

```
The average of these numbers is 3.500000
PS D:\MyData\Business\code playground\C++ course>
```

Copy

```
int main(){
    float a;
    a = funcAverage(5,2.8);
    printf("The average of these numbers is %f",a);
    return 0;
}
```

Copy

Code snippet: Calling the function by passing one integer and one float

```
The average of these numbers is 3.900000

PS D:\MyData\Business\code playground\C++ course>
```

Copy

And a general swap function named swapp for those variety of data types we have, would look something like the one below:

```
template <class T>

void swapp(T &a, T &b)

{

    T temp = a;

    a = b;

    b = temp;

}
```

Copy

So, this is how we utilize this powerful tool to avoid writing such overloaded codes. And this was all about function templates with single or multiple parameters. We covered them all in this tutorial.

Thank you, for being with me throughout, hope you liked the tutorial. If you haven't checked out the whole playlist yet, move on to codewithharry.com or my YouTube channel to access it. I hope you enjoy them all. We are now done with both class and function templates. See you all in the next tutorial where we'll see if a function template can be overloaded. Till then keep coding.

Member Function Templates & Overloading Template Functions in C++ | 68

So, since we have finished learning about the two template categories, we can now swiftly dive deep into if it's possible for a template function to get overloaded, and if yes, then how.

Before starting to know what an overloaded template function is, we'll learn how to declare a template function outside a using the scope resolution operator, '::'.

First, we'll revise how to write a function inside the class by just following the snippet given below.

1. We'll declare a template, then a class named Harry.
2. We'll then define a variable *data* inside that class with variable data type T.
3. We then make a constructor feeding the value received from the main to data.
4. And then, we'll write the function, *display* and write its code.

This was an unchallenging task. But when we need the function to be declared outside the class, we follow the code snippet 2.

```
template <class T>

class Harry

{

public:

    T data;

    Harry(T a)

    {

        data = a;

    }

    void display()

    {

        cout << data;

    }

};
```

Copy

Code Snippet 1: Writing function inside the class

Here, we first write the function declaration in the class itself. Then move to the outside and use the scope resolution operator before the function and after the name of the class Harry along with the data type T. We must specify the function data type, which is void here. And it must be preceded by the template declaration for class T.

And write the display code inside the function and this will behave as expected. See the output below the snippet.

```
template <class T>
class Harry
{
public:
    T data;
    Harry(T a)
    {
        data = a;
    }
    void display();
};

template <class T>
void Harry<T> :: display(){
    cout<<data;
}
```

Copy

Code Snippet 2: Writing function outside the class

So to check if it's working all fine, we'll call this function from the main.

```
int main()
{
    Harry<int> h(5.7);
    cout << h.data << endl;
    h.display();
    return 0;
}
```

Copy

Code Snippet 3: Calling the function from the main

And the output is:

```
5
5
PS D:\MyData\Business\code playground\C++ course>
```

Copy

Now, we'll move to the **overloading of a function template**. Overloading a function simply means assigning two or more functions with the same name, the same job, but with different parameters. For that, we'll declare a void function named func. And a template function with the same name. Follow the snippet below to do the same:

1. We made two void functions, one specified and one generic using a template.
2. The first one receives an integer and prints the integer with a different prefix.
3. The generic one receives the value as well as the data type and prints the value with a different prefix.
4. Now, we'll wish to see the output of the following functions, by calling them from the main. Refer to the main program below the snippet below.

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

```
void func(int a){
    cout<<"I am first func() "<<a<<endl;
}

template<class T>
void func(T a){
    cout<<"I am templatised func() "<<a<<endl;
}
```

Copy

Code Snippet 4: Overloading the template function

And now when we call the function func, we'll be interested to know which one among the two it calls. So here since we've entered a value with an integer parameter, it finds its exact match in the overloading and calls that itself. That is, it gives its exact match the highest priority. Refer to the output below the snippet:

```
int main()
{
    func(4);  //Exact match takes the highest priority
    return 0;
}
```

Copy

Code Snippet 5: Calling function func from the main

And the output is,

```
I am first func() 4
PS D:\MyData\Business\code playground\C++ course>
```

Copy

If we hadn't created the first function with int data type, the call would have gone to the templatised func only because a template function is an exact match for every kind of data type.

So this was enough preparation for the next topic, STL(Standard Template Library). It might have sounded boring to you for quite a few days, but the results would fascinate you once we enter STL, which is a must for all the competitive programmers out there.Thank you, for being with me throughout, hope you liked the tutorial. If you haven't checked out the whole playlist yet, move on to codewithharry.com or my YouTube channel to access it. I hope you enjoy them all. See you all in the next tutorial where we'll start the STL. Till then keep coding.

The C++ Standard Template Library (STL) | 69

We have been waiting so long to start this, but creating a base is as important as any other phase. So, today we'll be starting the most awaited topic, the STL(Standard Template Library).

There is a reason why I've been saying that this topic is a must for all the competitive programmers out there,so let's deal with that first.

Why is this important for competitive programmers?

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

An STL is a library of generic functions and classes which saves you time and energy which you would have spent constructing for your use. This helps you reuse these well tested classes and functions umpteenth number of times according to your own convenience.

To put this simply, STL is used because it is not a good idea to reinvent something which is already built and can be used to innovate things further. Suppose you go to a company who builds cars, they will not ask you to start from scratch, but to start from where it is left. This is the basic idea behind using STL.

COMPONENTS OF STL:

We have three components in STL:

1. Containers
2. Algorithm
3. Iterators

Let's deal with them individually;

Containers:

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

Algorithms:

Algorithms are a set of instructions which manipulates the input data to arrive at some desired result. In STL, we have already written algorithms, for example, to sort some data structure, or search some element in an array. These algorithms use template functions.

Iterators:

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

I'll give you a quick illustration of how they work combinedly.

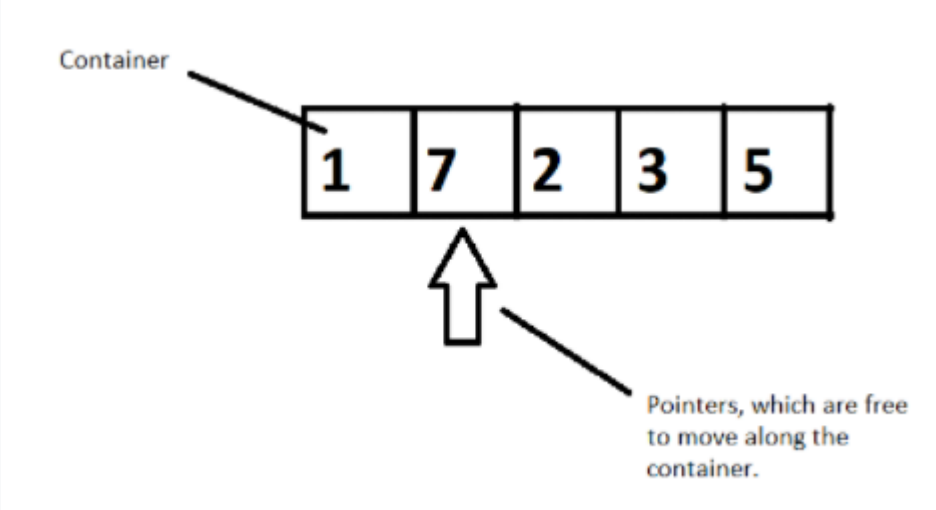


Figure 1: Illustration of how these three components work together

Suppose we have a container of integers, and we want to sort them in ascending order. We will have pointers which will help moving elements to places by pointing to it, following a well-constructed algorithm. So, here a container gets sorted by following an algorithm by the use of pointers. This is how they work in accordance with each other.

So, this was the basics of STL and the motivation behind using it in your programs. I hope I was able to introduce it to you.

Thank you, for being with me throughout, hope you liked the tutorial. If you haven't checked out the whole playlist yet, move on to codewithharry.com or my YouTube channel to access it. I hope you enjoy them all. See you all in the next tutorial where we'll dive deep in the containers and its different types. Till then keep coding.

Containers in C++ STL | 70

In the last tutorial, we had briefed about the three components of STL, namely,

Containers, objects which store data, **Algorithms,** set of procedures to process data, and **Iterators,** objects which point to some element in a container. Today, in this tutorial, we will be interested in discussing more about containers**.**

Containers are themselves of three types:

1. Sequence Containers
2. Associative Containers
3. Derived Containers

When we talked about containers, we said containers are objects which store data, but what are its three types all about? We'll discuss that too.

- **Sequence Containers**

A **sequence container** stores that data in a linear fashion. Refer to the illustration below to understand what storing something in a linear fashion means.

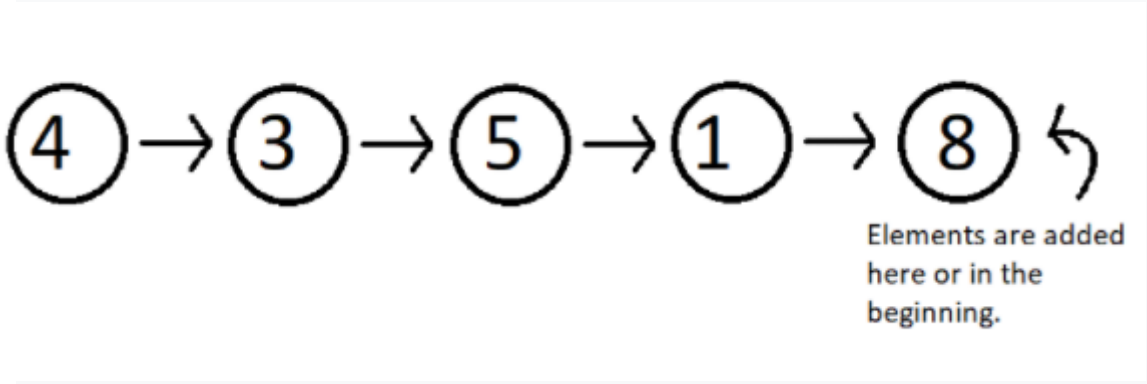


Figure 1: Elements stored in a linear fashion

Sequence containers include **Vector, List, Dequeue etc.** These are some of the most used sequence containers.

- **Associative Containers**

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

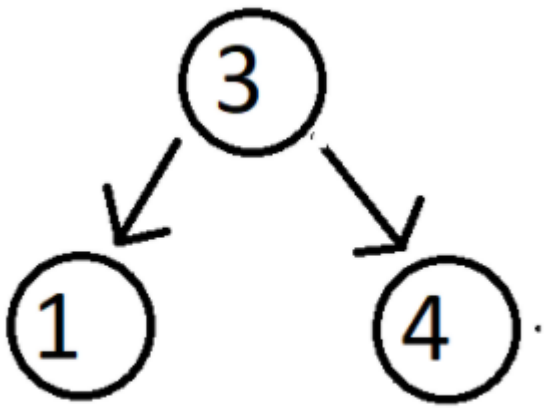


Figure 2: A tree-like structure

- **Derived Containers**

As the name suggests, these containers are derived from either the sequence or the associative containers. They often provide you with some better methods to deal with your data. They deal with real life modelling. Some examples of derived containers are **Stack, Queue, Priority Queue, etc.** The following illustration give you the idea of how a stack works.

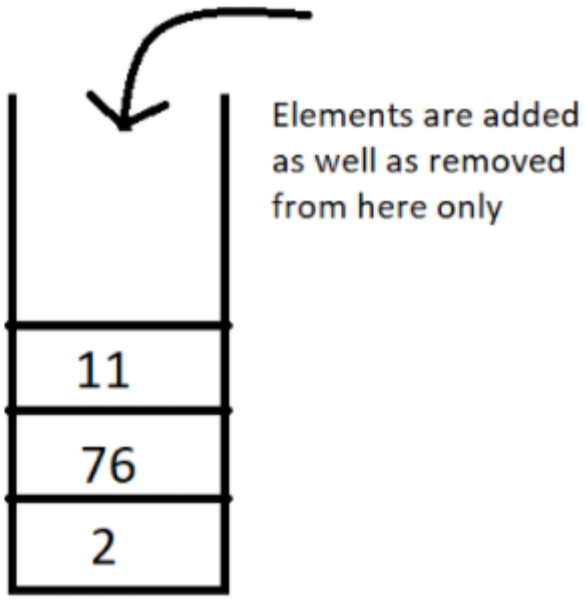


Figure 3: A stack, works on the first in first out [FIFO] method

Now since we have got the basic idea of all the three types of containers, a question which might arise is **when to use which**. So, let's deal with that,

In sequence containers, we have **Vectors**, which has following properties:

1. Faster random access to elements in comparison to array
2. Slower insertion and deletion at some random position, except at the end.
3. Faster insertion at the end.

In **Lists,** we have,

1. Random accessing elements is too slow, because every element is traversed using pointers.
2. Insertion and deletion at any position is relatively faster, because they only use pointers, which can easily be manipulated.

In associative containers, every operation except random access is faster in comparison to any other containers, be it inserting or deleting any element.

In associative containers, we cannot specifically tell which operation is faster or slower, we'll have to inspect every data structure separately, and to get a clearer picture of all of these, you can access my Data Structure course : Data Structures and Algorithms Course in Hindi

For now, I'd like to hold on to our topic STL, and get you a strong hold on this too. In the coming videos, we'll deal with our vectors, list, dequeues, set, multiset, maps, stack and much more. Just bear with me.

Thank you, for being with me throughout, hope you liked the tutorial. If you haven't checked out the whole playlist yet, move on to codewithharry.com or my YouTube channel to access it. I hope you enjoy them all. See you all in the next tutorial where we'll talk about Vectors in C++ STL in detail. Till then keep coding.

Vector In C++ STL | 71

In this video, we'll cover the Vectors in C++ STL. This is the tutorial we all were waiting for. Enough of the theory part. We'll go into our editors and code. So, to start, we'll have to include the header file <vector>. And the syntax we use to define a vector is:

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

Copy

Code Snippet 1: Syntax of declaring a vector

And suppose we want to have a vector of integers; the following program would do the needful:

```
include<iostream>
include<vector>


int main(){


    vector<int> vec1;
    return 0;
}
```

Copy

Code Snippet 2: Declaring a vector of integers

One benefit of using vectors, is that we can insert as many elements we want in a vector, without having to put some size parameter as in an array. In an array of 10 elements, for adding the 11th one, we'll have to make an array again.

Vectors provide certain methods to be used to access and utilise the elements of a vector, first one being, the push_back method. To access all the methods and member functions in detail, you can visit this site , std::vector - C++ Reference. This will be very handy and useful to you. I'll show you how some of them work in a program. Refer to the code snippet 3.

- **push_back() and size():**

1. First of all, don't forget to include the header file, <vector>.
2. Vectors have a method, push_back(), to insert elements in it from the rear end.
3. We'll define a variable, size, to store the size of the vector.
4. We'll then run a loop of size length, to receive the user input and push them back in the vector vec1.
5. We'll then call the display function.
6. We want to have a display function to display the contents of the vector. And pass reference of vec1 to the function.
7. We have another method size() which returns the size of the vector. We'll use this to traverse through all the elements and print them.
8. So, this is how a vector gets used.

```
include<iostream>
include<vector>
```

```
using namespace std;
void display(vector<int> &v){
    for (int i = 0; i < v.size(); i++)
    {
        cout<<v[i]<<" ";
    }
    cout<<endl;
}
int main(){
    vector<int> vec1;
    int element, size;
    cout<<"Enter the size of your vector"<<endl;
    cin>>size;
    for (int i = 0; i < size; i++)
    {
        cout<<"Enter an element to add to this vector: ";
        cin>>element;
        vec1.push_back(element);
    }
    display(vec1);
    return 0;
}
```

Copy

Code Snippet 3: A program to demonstrate the use of push_back and size methods

The output of the above program:

```
Enter the size of your vector
3
Enter an element to add to this vector: 5
Enter an element to add to this vector: 3
Enter an element to add to this vector: 7
5 3 7
PS D:\MyData\Business\code playground\C++ course>
```

Copy

Figure 1: Output of the above program

Similarly, we can even build float vectors, and we can even templatise the display function.

- **pop_back():**

This method of vectors, deletes the last element of the vector. Refer to the code snippet and the following output below.

```
    display(vec1);
    vec1.pop_back();
    display(vec1);
```

Copy

Code Snippet 4: Using pop_back in a vector

So, now you can see how this method deleted the last element 7 from the vector.

```
Enter the size of your vector
3
Enter an element to add to this vector: 5
Enter an element to add to this vector: 3
Enter an element to add to this vector: 7
5 3 7
5 3
PS D:\MyData\Business\code playground\C++ course>
```

Copy

Figure 2: Output of the above program

- **Insert (**iterator, element to insert**):**

This method of vectors inserts an element to the position the iterator is pointing to. Now how to evoke that iterator? Refer to the snippet and the output below:

We can generate an iterator using the scope resolution iterator by the following syntax:

```
  vector<int> :: iterator iter = vec1.begin();
```

Copy

Code Snippet 5: Declaring a vector iterator

Using **begin ()** points the iterator to the starting of the vector. We can now increment the pointer according to our choice and insert any element at that position.

```
display(vec1);
    vector<int> :: iterator iter = vec1.begin();
    vec1.insert(iter,566);
    display(vec1);
```

Copy

Code Snippet 6: Demonstrating an insert method

The output of the above program is:

```
Enter the size of your vector
3
Enter an element to add to this vector: 5
Enter an element to add to this vector: 3
Enter an element to add to this vector: 7
5 3 7
566 5 3 7
PS D:\MyData\Business\code playground\C++ course>
```

Copy

Figure 3: Output of the above program

Similarly, **v.at(i)** can be used instead of **v[i]**. They will work the same.

We have different ways to declare a vector. I'll list some of them through the snippet below.

1. First one is a vector with no length and elements specified.
2. Second one is a vector of length 4 and no elements.
3. Third one is a vector made from the second one.
4. And last one, is a vector with length 6 and all the elements being 3.

```
vector<int> vec1;      //zero length integer vector
vector<char> vec2(4);  //4-element character vector
vector<char> vec3(vec2);//4-element character vector from vec2
vector<int> vec4(6,3); //6-element vector of 3s
```

Copy

Code Snippet 7: Demonstrating different ways to declare a vector

So this was all the basics of a vector, and enough for you to get started using it. With this I'll finish today's tutorial. I'll recommend you all to visit my DSA playlist, Data Structures and Algorithms Course in Hindi to get acquainted with all the data structures and more.

Thank you, for being with me throughout, hope you liked the tutorial. If you haven't checked out the whole playlist yet, move on to codewithharry.com or my YouTube channel to access it. I hope you enjoy them all. See you all in the next tutorial where we'll learn about Lists in C++ STL. Till then keep coding.

List In C++ STL | 72

Before this tutorial, we covered templates, STL, and the last video was an efficient introduction to the vectors. Today, we'll learn about Lists in C++ STL.

A List is a bi-directional linear storage of elements. Few key features as to why a list should be used is,

1. It gives faster insertion and deletion operations.
2. Its access to random elements is slow.

What makes a list different from an array?

An array stores the elements in a contiguous manner in which inserting some element calls for a shift of other elements, which is time taking. But in a list, we can simply change the address the pointer is pointing to. I'll show you how these work via an illustration.

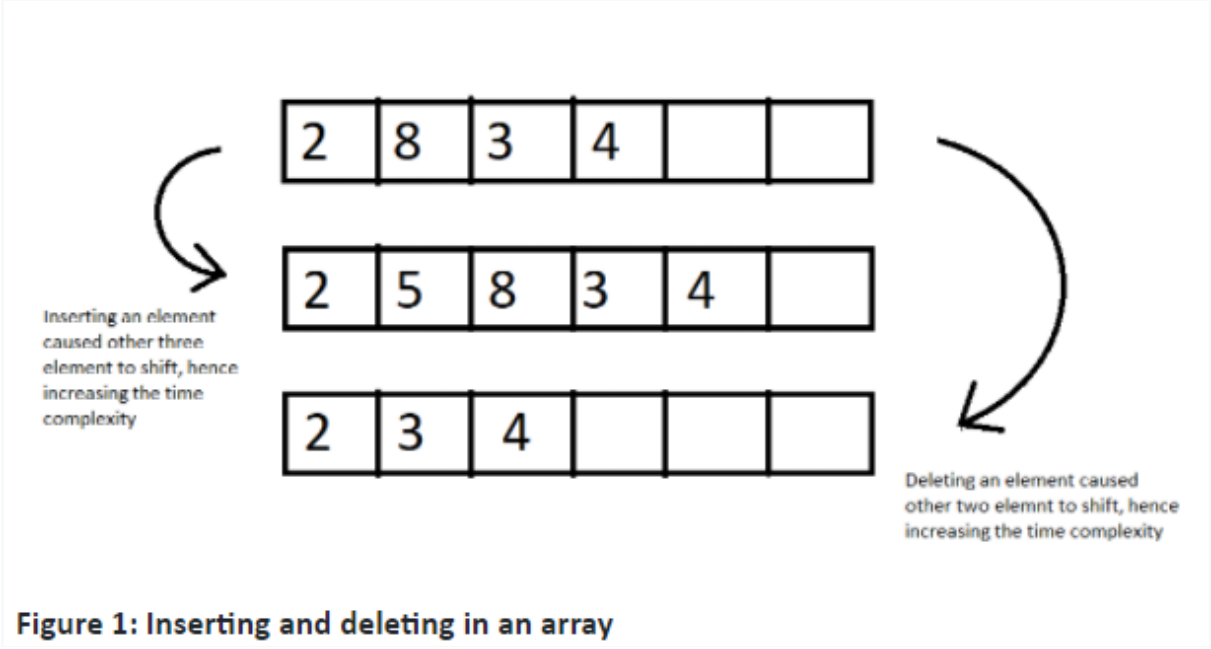


Figure 1: Inserting and deleting in an array

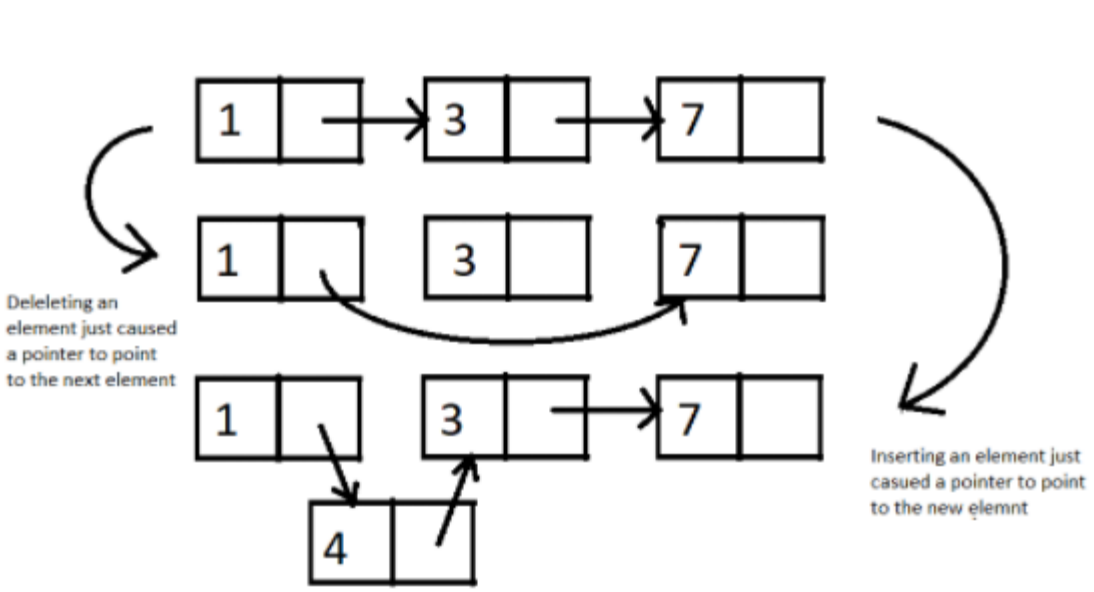


Figure 2: Insertion and deletion in a list

Let's move on to our editors and write some code using lists and its methods.

Understanding code snippet 2:

- Before using lists, we must include the header file <list>.
- Using a simple program, we'll iterate through the list and display its contents.
- As we did for vectors, first define a list list1.
- And push_back a few elements, and pass the list to a display function via reference.
- Due to the fact that a list element cannot be directly accessed by its index, we must traverse through each element and print them.
- We define a list iterator using this syntax:

```
    list<int> :: iterator it;
```

Copy

Code Snippet 1: Syntax for defining a list iterator

- We use two methods, **begin()** and **end()** to define the starting and the end of the loop. **end()** returns the pointer next to the last element.
- We dereference the list iterator, using * to print the element at that index.

```
include<iostream>
include<list>


using namespace std;


void display(list<int> &lst){
    list<int> :: iterator it;
    for (it = lst.begin(); it != lst.end(); it++)
    {
        cout<<*it<<" ";
    }


}
```

```
int main(){

    list<int> list1;  //empty list of 0 length

    list1.push_back(5);
    list1.push_back(7);
    list1.push_back(1);
    list1.push_back(9);
    list1.push_back(12);

    display(list1);

    return 0;
}
```

Copy

Code Snippet 2: A program using list

```
5 7 1 9 12
PS D:\MyData\Business\code playground\C++ course>
```

Copy

Figure 3: Output of the above program

We can also enter elements in a list using the iterator and its dereferencer. See the snippet below.

```
int main(){

    list<int> list2(3);  //empty list of length 3
    list<int> :: iterator it = list2.begin();
    *it = 45;
    it++;
    *it = 6;
    it++;
    *it = 9;
    it++;

    display(list2);

    return 0;
}
```

Copy

Code Snippet 3: Inserting in list using its iterator

```
45 6 9
PS D:\MyData\Business\code playground\C++ course>
```

Copy

Figure 4: Output of the above program

- **Using pop_back() and pop_front():**

We can use pop_back() to delete one element from the back of the list everytime we call this method and pop_front() to delete elements from the front. These commands decrease the size of the list by 1. Let me show you how these work by using them for list1 we made.

```
    list1.pop_back();
    display(list1);
    list1.pop_front();
    display(list1);
```

Copy

Code Snippet 4: Using pop_back and pop_front in list

The output of the above program is:

```
5 7 1 9
7 1 9
PS D:\MyData\Business\code playground\C++ course>
```

Copy

Figure 5: Output of the above program

- **Using remove():**

We can remove an element from a list by passing it in the list remove method. It will delete all the occurrences of that element. The remove method receives one value as a parameter and removes all the elements which match this parameter. Refer to the use of remove in the below snippet.

```
int main(){


    list<int> list1;  //empty list of 0 length


    list1.push_back(5);
    list1.push_back(7);
    list1.push_back(1);
    list1.push_back(9);
    list1.push_back(9);
    list1.push_back(12);

    list1.remove(9);
    display(list1);


    return 0;
}
```

Copy

Code Snippet 5: Deleting elements in list using remove()

The output of the above program is:

```
5 7 1 12
PS D:\MyData\Business\code playground\C++ course>
```

Copy

Figure 6: Output of the above program

- **Using sort():**

We can sort a list in ascending order using its sort method. Look for the demo below.

```
    display(list1);
    list1.sort();
    display(list1);
```

Copy

Code Snippet 6: Sorting elements in list using sort()

```
5 7 1  9 12
1 5 7  9 12
PS D:\MyData\Business\code playground\C++ course>
```

Copy

Figure 7: Output of the above program

I consider this much to be enough for lists. There are still a lot of them, but you will require no more, and even if you feel like exploring more, move onto std::list - C++ Reference and read about all the lists methods. This was all from my side. For more information on linked lists, visit my DSA playlist, Data Structures and Algorithms Course in Hindi.

Thank you, for being with me throughout, hope you liked the tutorial. If you haven't checked out the whole playlist yet, move on to codewithharry.com or my YouTube channel to access it. I hope you enjoy them all. See you all in the next tutorial where we'll learn about Maps in C++ STL. Till then keep coding.

Map In C++ STL | 73

So far, we have learned about vectors and lists in C++ STL, and today we will be learning about maps in C++ STL. It is important to clarify that whatever I have taught and whatever I will be teaching in the coming tutorials about STL isn't everything. And it is definitely not all. These are just the most important STL containers we will use. You have already seen how to explore more about STL from C++ - Containers.

We will now discuss maps, and because it is impractical to have every method on our fingers, I'd ask you all to also refer to the following website std::map - C++ Reference

A map in C++ STL is an associative container which stores key value pairs. To elaborate, a map stores a key of some data type and its corresponding values of some data type. For example: a teacher wants to store the marks of students which in future can be accessed by their names. Here, keys are the student names, and their marks are the corresponding values. Refer to the illustration below:

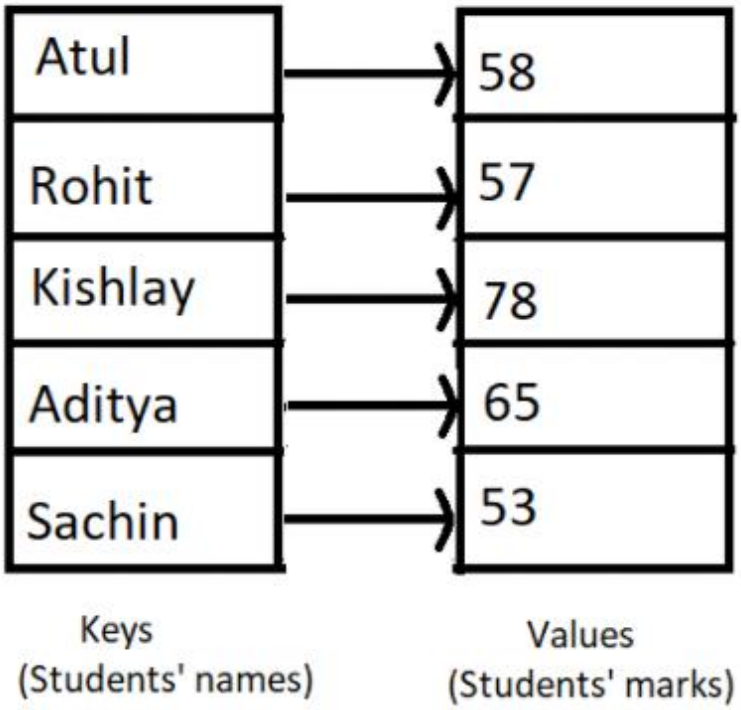


Figure 1: Illustrative diagram of a key value pairs

We can now shift to our editors and see how maps can be used in C++. Don't forget to include the header file <map>.

The syntax for declaring a map is:

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

Copy

Code Snippet 1: Syntax for declaring a map

And we can now write the program for storing the key value pairs of students' names and students' marks keeping in mind the illustration above. Refer to the snippet below.

Understanding code snippet 2:

1. Include the header file map and string(if using string methods).
2. Let's create a map in which the key is a string (names) and the values are integers (marks), and we'll call it marksMap.
3. And to assign some key a value, we use the index method. Here the index of a map element will be the students' name and the value will be the marks.
4. Make some 4-5 elements.
5. Identify the iterator of this map by using the scope resolution operator.
6. Loop through the map elements using two map methods; **begin()** to point at the beginning of the map, and **end()** to point next to the last element of the map.
7. While we loop through the map, we use the dereference operator * to fetch the element present where the pointer is pointing to. And since a map stores element in a key value pair, we can use its first and second method to access the keys and the values respectively. **.first** accesses the first value of a pair that is our map key here, and **.second** accesses the second value of the pair that is our map values here.
8. There is one thing to keep in mind: Maps always sort these pairs by the key elements. You can review the output of the following snippet to see how these pairs are sorted.

```
include<iostream>

include<map>

include<string>
```

```
using namespace std;

int main(){

    // Map is an associative array
    map<string, int>  marksMap;
    marksMap["Atul"] = 58;
    marksMap["Rohit"] = 57;
    marksMap["Kishlay"] = 78;
    marksMap["Aditya"] = 65;
    marksMap["Sachin"] = 53;

    map<string,int> :: iterator iter;
    for (iter = marksMap.begin(); iter != marksMap.end(); iter++)
    {
        cout<<(*iter).first<<" "<<(*iter).second<<"\n";
    }


    return 0;
}
```

Copy

Code Snippet 2: A program to store names and marks using map

As you can see, the map pairs got sorted according to its key.

```
Aditya 65
Atul 58
Kishlay 78
Rohit 57
Sachin 53
PS D:\MyData\Business\code playground\C++ course>
```

Copy

Figure 2: Output of the above program

We have one more method to insert elements in a map. We can use **.insert()**

Syntax for using .insert is:

```
marksMap.insert({pair_1,pair_2.....pair_n});
```

Copy

Code Snippet 3: Syntax for inserting pairs in map

We will insert some elements into our map in snippet 2 by using the insert method.

```
 marksMap.insert( { {"Rohan", 89}, {"Akshat", 46} } );
```

Copy

Code Snippet 4: Program to insert two pairs in a map.

We can see the output to check if the above two pairs got inserted into the map or not.

```
Aditya 65
Akshat 46
Atul 58
Kishlay 78
Rohan 89
Rohit 57
Sachin 53
PS D:\MyData\Business\code playground\C++ course>
```

Copy

Figure 3: Output of the above program

So, yes, it worked. And the output was correct.

And I'd ask you all to explore more of these methods from the links I gave you above and start writing codes using them. This is the best way to learn. For example, you can use the **size()** method to get the size of the map container , **empty()** method to check if the map container is empty or not, and it returns a boolean. Therefore, this shouldn't be a big deal for you.

Thank you for being with me throughout the tutorial. I hope you enjoyed it. If you haven't checked out the whole playlist yet, move on to codewithharry.com or my YouTube channel to access it. You will surely enjoy them all. See you all in the next tutorial where we'll learn about Maps in C++ STL. Till then keep coding.

unction Objects (Functors) In C++ STL | 74

In the last tutorial we completed learning about some of the most commonly used containers, vector, list, map and their methods. Today we'll start with function objects in C++ STL.

What is a function object?

A function object is a function wrapped in a class so that it is available as an object.

That is, we can then use a function as an object. The question that might have been raised in your mind would be, **why to substitute a function with an object**? The answer is to make them all usable in an Object-Oriented Programming paradigm. Now what does that mean? We'll try decoding the purpose of using functions as an object via a program. So, hold onto your editors.

Understanding code snippet 1:

- Be sure to include the header file < functional> before you do anything else.
- And let's create an array of some 6 elements.
- Suppose we want to sort this array in ascending order. So we'll include a header file <algorithm> and write the syntax of the sort object which is,

```
sort(address of first element, address of last element);
```

Copy

Code Snippet 1: Syntax for sort algorithm

- And let's just sort from the beginning to the 5th element.
- And run a loop to see the resultant array.

```
include<iostream>
include<functional>
include<algorithm>

using namespace std;

int main(){
```

```
    // Function Objects (Functor) : A function wrapped in a class so that it is available like an object
    int arr[] = {1, 73, 4, 2, 54, 7};
    sort(arr,arr+5);
    for (int i = 0; i < 6; i++)
    {
        cout<<arr[i]<<endl;
    }

    return 0;
}
```

Copy

Code Snippet 2: Program to sort an array in ascending order

Output of the above program is given below. And you'll notice that the last element remained untouched.

```
1
2
4
54
73
7
PS D:\MyData\Business\code playground\C++ course>
```

Copy

Figure 1: Output of the above program

But what if we wanted to sort the same array in descending order, since the sort function can default sort in ascending order only? So, here comes our saviour, **functional objects**. Our sort function also takes a third parameter which is a functor (functional object).

Let's see how they work via the snippet below:

- Among all the different functors we have, the one to help this sort function to sort the array in descending order, is the **greater<int>()**.

```
sort( arr, arr+6, greater< int >( ));
```

Copy

Code Snippet 1: Syntax for using a functor in an algorithm

- And that's it. Our array will now get sorted in descending order.

See the output after the above changes we made:

```
73
54
7
4
2
1
PS D:\MyData\Business\code playground\C++ course>
```

Copy

Figure 1: Output of the above program after using a functor greater<int>()

It would be unnecessarily lengthy to review all the functors, as my role was to introduce them to you and show you how they are used.

In addition, I invite you all to explore the other function objects on the site Function objects. You should go through them lightly because a lot of them would be overwhelming to you as a beginner. We use most of these functors for our STL algorithms, so you might want to check all those algorithms on Functions in <algorithm> - C++ Reference. You can go through them at your own pace. Incorporate them into your programs. Take advantage of them and use them how you see fit.

Thank you for being with me throughout the tutorial. I hope you enjoyed it. If you haven't checked out the whole playlist yet, move on to codewithharry.com or my YouTube channel to access it. You will surely enjoy them all. Looking forward to seeing you all next time. Till then keep coding.