Object Oriented Programming Using C++

Basic Structure of C++

```
#include<iostream>

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

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 quotations " " 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.

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

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

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.

- Variable Scope
- Data Types

Before explaining the concept of variable scope, I would like to clarify about variables a little more. Variables 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.

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

```
# 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;
}
```

Basic Input/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 example 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)

```
# 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;
}
```

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

Header files & Operators

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 ship 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

Operator	Operation
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulo Operation (Remainder after division)
++	Increment by 1
--	Decrement by 1

Arithmetic operators are used for performing mathematical operations like (+, -, *). The 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.

Assignment Operators

Operator	Example	Equivalent to
=	**a = b;**	**a = b;**
+=	**a += b;**	**a = a + b;**
-=	**a -= b;**	**a = a - b;**
***=**	**a *= b;**	**a = a * b;**

/=	**a /= b;**	**a = a / b;**
%=	**a %= b;**	**a = a % b;**

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

- **Comparison Operators**

Operator	Meaning	Example
==	Is Equal To	3 == 5 gives us false
!=	Not Equal To	3 != 5 gives us true
>	Greater Than	3 > 5 gives us false
<	Less Than	3 < 5 gives us true
>=	Greater Than or Equal To	3 >= 5 give us false
<=	Less Than or Equal To	3 <= 5 gives us true

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.

- **Logical Operators**

Operator	Example	Meaning
&&	expression1 **&&** expression2	Logical AND. True only if all the operands are true.
||	expression1 || expression2	Logical OR. True if at least one of the operands is true.
!	**!**expression	Logical NOT. True only if the operand is false.

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

```
// 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;
```

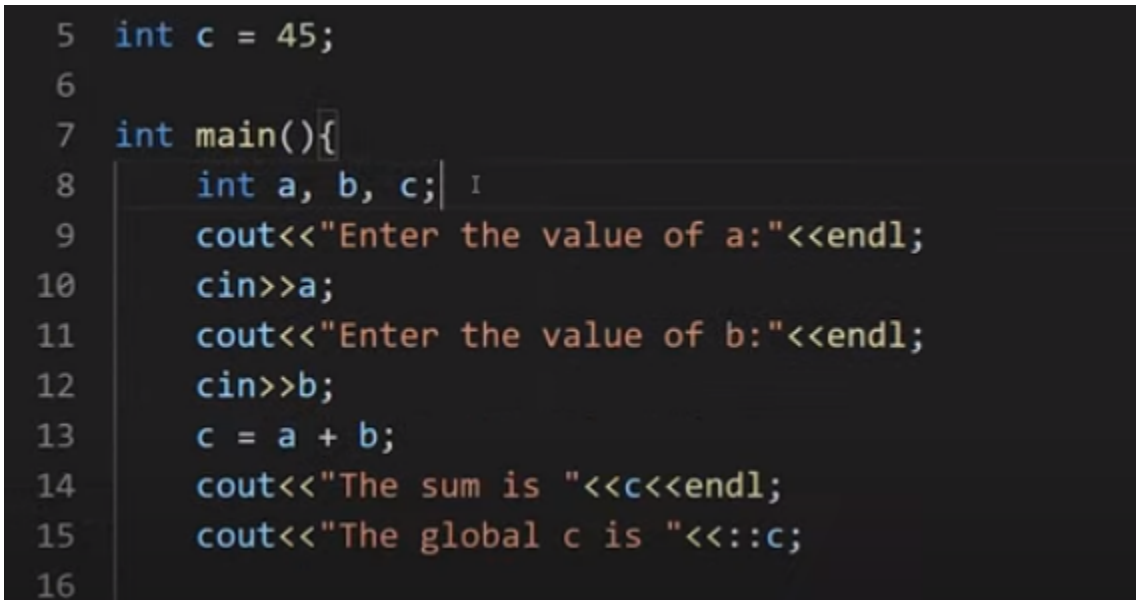
```
    cout<<"The value of a - b is "<<a-b<<endl;
    cout<<"The value of a * b is "<<a*b<<endl;
    cout<<"The value of a / b is "<<a/b<<endl;
    cout<<"The value of a % b is "<<a%b<<endl;
    cout<<"The value of a++ is "<<a++<<endl;
    cout<<"The value of a-- is "<<a--<<endl;
    cout<<"The value 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;
}
```

Reference Variables & Typecasting

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

Built-in Data Types

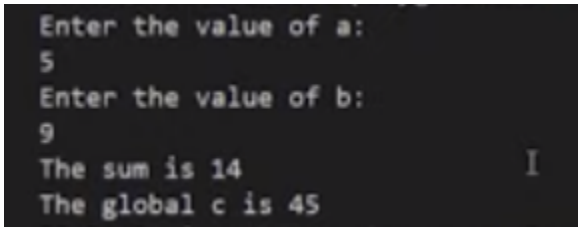
As discussed in our previous lectures, built-in data types are predefined by the language and can be used directly. An example program for built-in data types is shown in figure 1.



```
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.

```
The size of 34.4 is 8
The size of 34.4f is 4
The size of 34.4F is 4
The size of 34.4l is 12
The size of 34.4L is 12
The value of d is 34.4
The value of e is 34.4
```

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 aliases. 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.

```
float x = 455;
float & y = x;
cout<<x<<endl;
cout<<y<<endl;
```

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 a 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.

```
PS D:\Business\code playground\C++ course> c
455
455
PS D:\Business\code playground\C++ course>
```

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

```
#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;
}
```

Constants, Manipulators & Operator Precedence

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

Constants in C++

Syntax:-

const data_type constant_name = value

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.

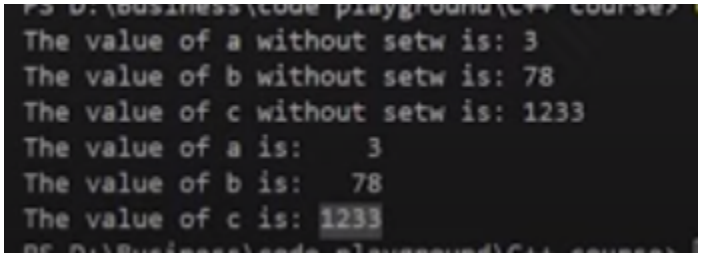


Figure 4: Manipulators Program Output

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.

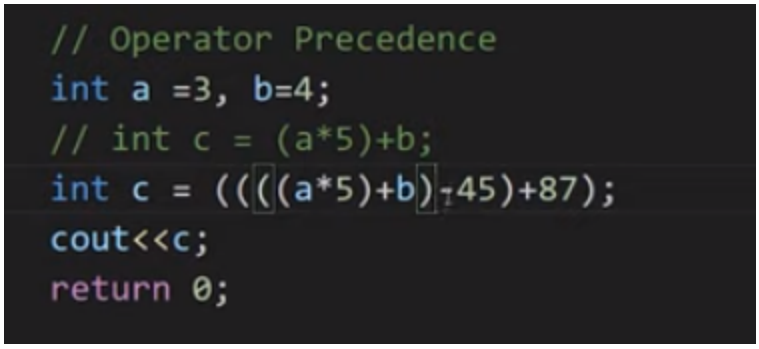


Figure 5: Operator Precedence & Associativity Example program

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.

```
#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;
}
```

Control Structures

- 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 a program executes instructions one after another. An example diagram for the sequence structure is shown in figure 1.

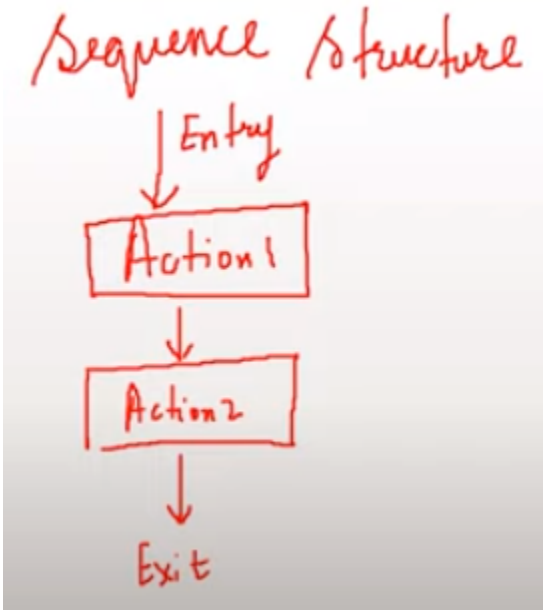


Figure 1: Sequence Structure

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.

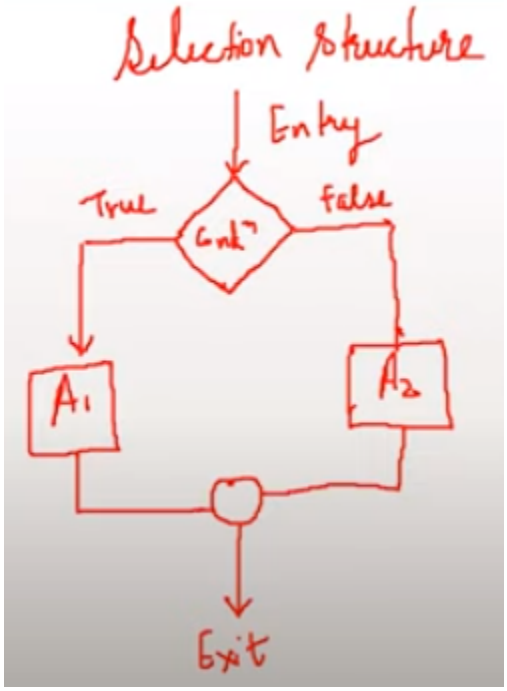


Figure 2: Selection Structure

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.

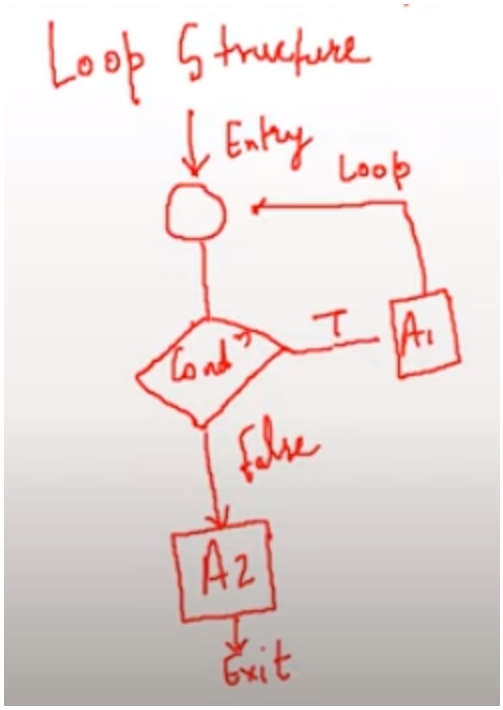


Figure 3: Loop Structure

If Else Statements in C++

Syntax:-

```
if(condition)
{
   // Statements to execute if
   // condition is true
}
-------------------------------------------------------------------------------------------------------------
if (condition)
{
    // Executes this block if
    // condition is true
}
else
{
    // Executes this block if
    // condition is false
}
```

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 the “cin" function to get 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 playgroun
Tell me your age
81
You can come to the party
```

Figure 5: If-Else Program Output

As can be seen in figure 5, 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++

```
switch (n)
{
   case 1: // code to be executed if n = 1;
      break;
   case 2: // code to be executed if n = 2;
      break;
   default: // code to be executed if n doesn't match any cases
}
```

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” skipped all the other cases due to the “break” statement and printed “Done with switch case” which was outside of the switch case statement.

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

int main(){

    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;
}
```

loops

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 (Initialization; condition; increment/decrement) {
  // code block to be executed
}
```

For loop helps 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.

Figure 2: For Loop Program Output

While Loop in C++

```
Initialization;
while (condition) {

  // code block to be executed
increment/decrement;
}
```

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.

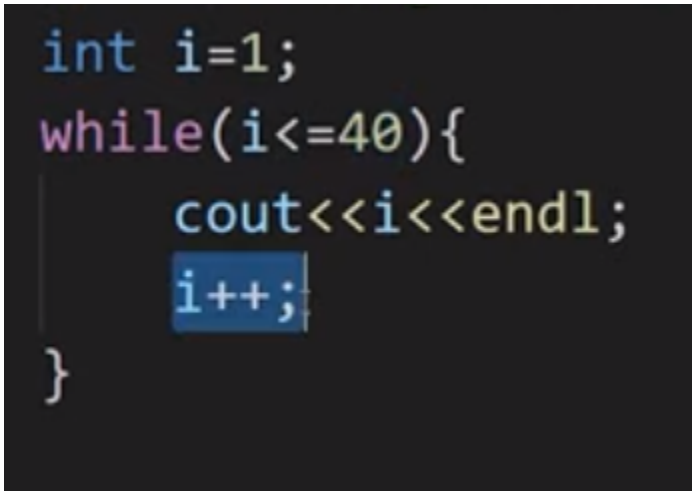


Figure 3: While Loop Program

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++

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

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.

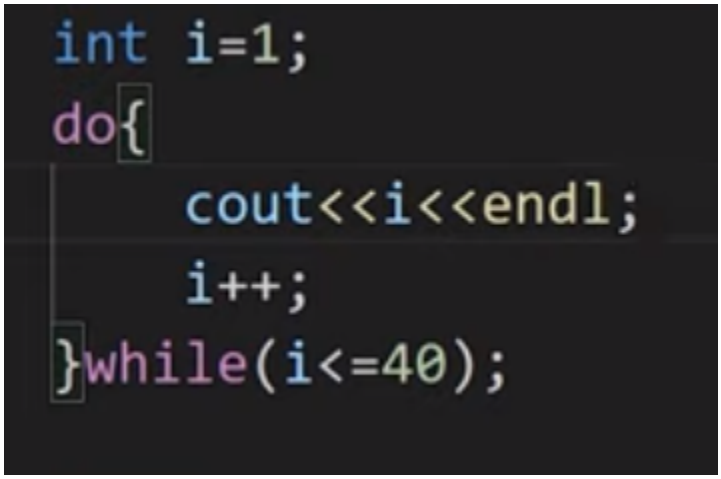


Figure 4: Do-While Loop Program

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

```
#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;
}
```

Break and Continue Statements

- 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.

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.

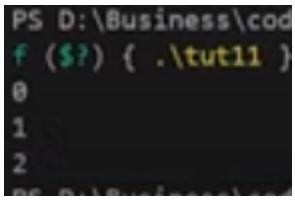


Figure 2: Break Statement Program output

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.

```
#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;
}
```

Pointers

Pointers in C++

A pointer is a data type which holds the address of another 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.

```
Data_Type *pointer_name;
```

```
int a=3;
int* b = &a;
cout<<b;
```

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.

```
PS D:\Business\code pla
0x61ff08
PS D:\Business\code pla
```

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.

```
int a=3;
int* b = &a;
cout<<"The address of a is "<<&a<<endl;
cout<<"The address of a is "<<b<<endl;
```

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.

```
The address of b is 0x61ff04
The address of b is 0x61ff04
The value at address c is 0x61ff08
The value at address value_at(value_at(c)) is 3
```

Figure 8: Pointer to Pointer Example Program Output

```
#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

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 variables 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];

```
Data_type array_name[length];
```

An example program for an array is shown in the 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;s
```

Copy

Code Snippet 1: Array Program 1

As shown in the code snippet, we initialize 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.

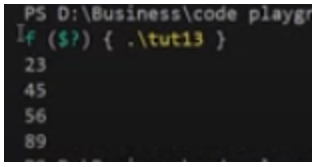


Figure 1: Array Program 1 Output

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.

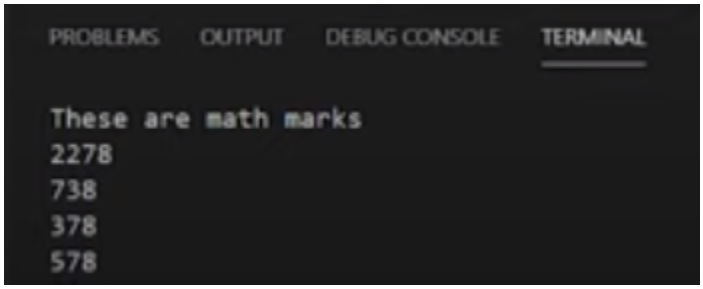


Figure 2: Array Program 2 Output

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.

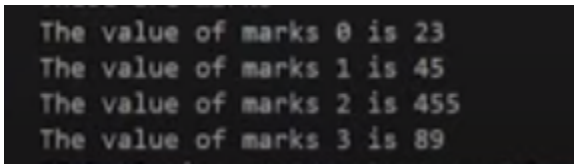


Figure 3: Array program with loop output

Pointers and Arrays

Storing the address of an array into a pointer is different from 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.

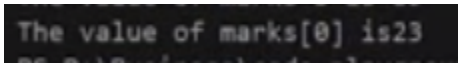


Figure 4: Pointer and Array Program Output

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.

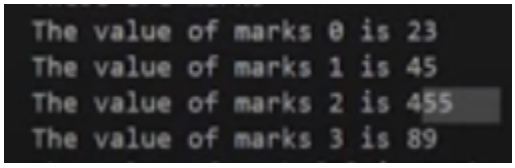


Figure 5: Pointer and Array Program 2 Output

```
#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;
    // cout<<marks[3]<<endl;

    for (int i = 0; i < 4; i++)
```

```
    {
        cout<<"The value of marks "<<i<<" is "<<marks[i]<<endl;
    }

    // Quick quiz: do the same using while and do-while loops?

    // Pointers and arrays
    int* p = marks;
    cout<<*(p++)<<endl;
    cout<<*(++p)<<endl;
    // 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;

    return 0;
}
```

Structures, Unions & Enums

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.

Syntax:-

```
struct structureName{
    member1;
    member2;
    member3;
    .
    .
     memberN;
};
```

Eg:-

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

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 structured employees is shown in Code Snippet 2.

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

As shown in Code Snippet 2, 1st we have created a structure variable “paddy” 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;
```

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 paddy;
    struct employee shubham;
    struct employee rohanDas;
    paddy.eId = 1;
    paddy.favChar = 'c';
    paddy.salary = 120000000;
    cout<<"The value is "<<paddy.eId<<endl;
    cout<<"The value is "<<paddy.favChar<<endl;
    cout<<"The value is "<<paddy.salary<<endl;
    return 0;
}
```

As shown in Code Snippet 4, we have created a structure instance “paddy” 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.

Syntax-	Eg:-

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

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;
}
```

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.

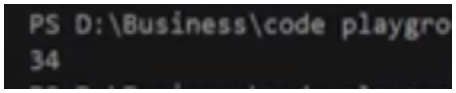


Figure 1: Creating Union Instance Output

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.

```
Syntax:-
enum enumerated-type-name{value1, value2, value3…..valueN};
int main(){
    enum Meal{ breakfast, lunch, dinner};
    Meal m1 = lunch;
    cout<<m1;
    return 0;
}
```

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.

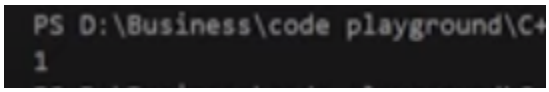


Figure 2: Enums Program Output

```
#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 paddy;
     struct employee shubham;
     struct employee rohanDas;
     paddy.eId = 1;
     paddy.favChar = 'c';
     paddy.salary = 120000000;
     cout<<"The value is "<<paddy.eId<<endl;
     cout<<"The value is "<<paddy.favChar<<endl;
     cout<<"The value is "<<paddy.salary<<endl;
    return 0;
}
```

Functions & Function Prototypes

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.

```
Syntax:-
Type function_name(type arg1,type arg2){
//Code
}
```

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

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;
}
```

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.

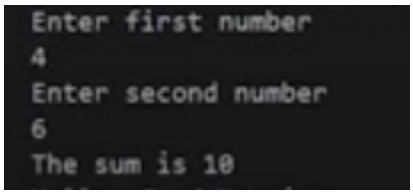


Figure 1: Function Output

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.

Syntax:-
 	
type function_name(arg1,arg2);
	Eg:0
	// Function prototype
	int sum(int a, int b);

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";
}
```

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”

```
#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

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
}
```

As shown in Code Snippet 1, we created a swap function which is taking two parameters “int a” and “int b”. In the 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;
}
```

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.

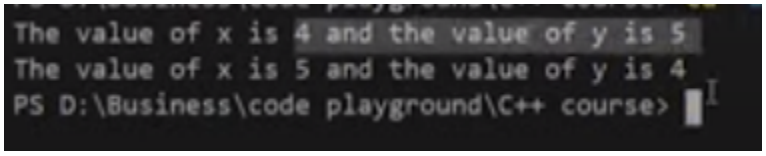


Figure 1: Call by Value Swap Function Output

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.

Syntax:-
 	
Type function_name(type *ptr1,type *ptr2){
//Code
}
	// 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
	}

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;
}
```

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.

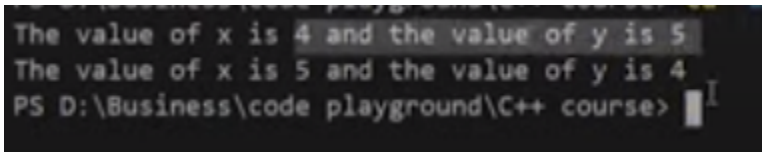


Figure 2: Call by Pointer Swap Function Output

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.

```
Syntax:-
Type function_name(type &arg1,type &arg2){

//Code
}
```

```
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
}
```

As shown in Code Snippet 5, we created a swap function that is taking reference of “int &a” and “int &b” as parameters. In the 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.

```
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
}
```

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.

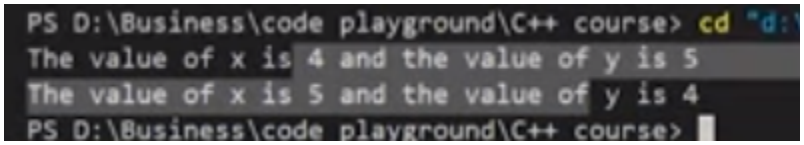


Figure 3: Call by Reference Swap Function Output

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.

```
#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

Inline Functions in C++

Inline functions are used to reduce the function call. When one function is being called multiple times in the program it increases the execution time, so an 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.

Syntax:-
Inline type function_name(type arg1,type arg2){
}
	inline int product(int a, int b){
	  return a*b;
	}

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;
}
```

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.

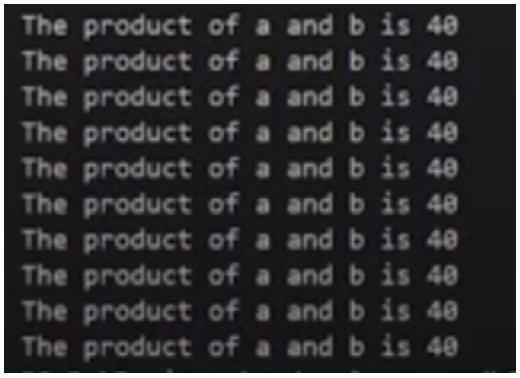


Figure 1: Inline Function Output

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

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.

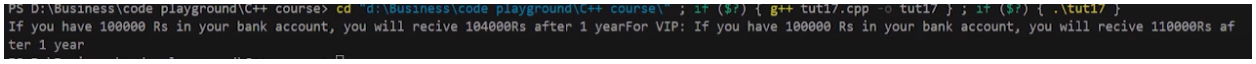


Figure 2: Default Argument Example Program Output

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.

Syntax:-
Type function_name(const type arg1){
}
	int strlen(const char *p){
	
	}

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

```
#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;
}
```

Recursions & Recursive

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 a recursive function otherwise the 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);
}
```

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

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.

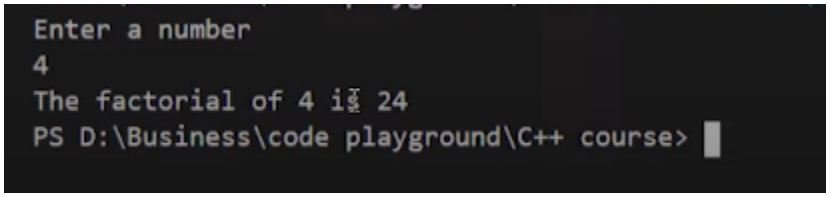


Figure 1: Factorial Recursive Function Output

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);
}
```

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;
}
```

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.

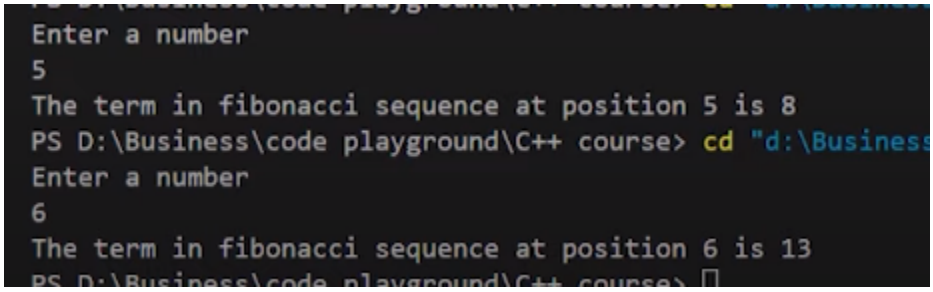


Figure 2: Fibonacci Recursive Function Output

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.

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

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

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



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;
}
```

Function Overloading

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.

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” functions is shown in Code Snippet 2.

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 the program will not run.

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

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.

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.

```
The volume of cuboid of 3, 7 and 6 is 126
The volume of cylinder of radius 3 and height 6 is 169
The volume of cube of side 3 27
PS D:\Business\code playground\C++ course>
```

Figure 2: Volume Function Output

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

```
#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

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

Classes, Public and Private access modifiers

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.

(1) Class is a way to bind data with its associated function together iit allow user to hind data from external use.

1 Class Declaration

2 Class Function Definitions

Syntax:-	Eg:-

class class_name{
private: // No entry to private class
variable declaration;
function declaration;
public: // Allow for public area
variable declaration;
function declaration;
};
	class item{
	int number;
	float cost;
	public:
	void getdata(int a,int b);
	void putdata(void);
	};

(2) Creating Object:-

Class specifies what it contains. We can create objects after class declaration.

Syntax:-	Eg:-
class_name object_name;	item i1;

(3) Accessing Class Member

syntax:-	Eg:-
object_name.function_name(argumentlist);	i1.getdata(100,75.5);

(4) Defining Member function

-Inside Class

-Outside Class

(i)Outside Class

Declared member function should be define outside class separately

syntax:-	Eg:-

return-type class_name::function_name(argument declaration){
function body;
}
	void item :: getdata(int a,int b){
	number =a;
	cost = b;
	}

-Inside Function Definition

```
class item{
        int number;
        int cost;
        public:
        void getdata(int a,float b); //Inline //Declaration
        void putdata(void){ //Definition inside the class
        cout<<number<<endl;
        cout<<cost<<endl;
        }
};
```

Public Access Modifier in C++

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

Private Access Modifier in C++

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

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

```
#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 Pradip;
        // Pradip.a = 134; -->This will throw error as a is private
        Praidp.d = 34;
        Praidp.e = 89;
        pradip.setData(1,2,4);
        Pradip.getData();
        return 0;
}
```

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 features than structures such as; classes that can have methods and properties.

- Classes have a feature to make class members public and private.
- In C++ objects can be declared along with class deceleration as shown in Code Snippet 1.

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

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);
};
```

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;
}
```

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);
        }
        }
}
```

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';
        }
        }
}
```

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;
}
```

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;
}
```

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.

```
// OOPs - Classes and objects

// C++ --> initially called --> C with classes by stroustrup
// 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 declaration like this:
/* class Employee{
```

```
        // Class definition
        } paddy, rohan, lovish; */
// paddy.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;
}
```

An array of Objects

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;
        }
};
```

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;
}
```

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.

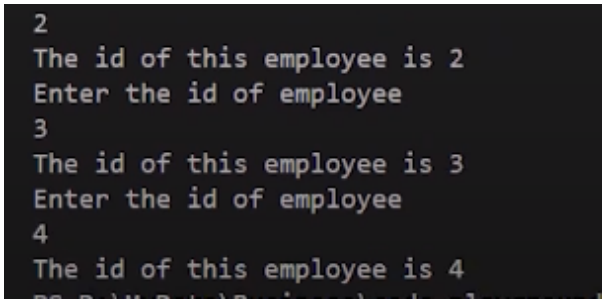


Figure 1: Employee Program Output

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 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;
        }
};
```

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.

```
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;
        }
};
```

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.

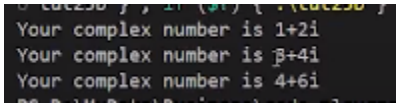


Figure 2: Complex Program Output

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

        return 0;
}
```

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

        return 0;
}
```

Friend Functions

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.

```
Syntax:-
friend class_name fuction_name(class_name object_name1,class_name object_name2 ){
//Code
}
```

```
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 function 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;
}
```

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;
}
```

As shown in Code Snippet 2:

- We have created objects “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.

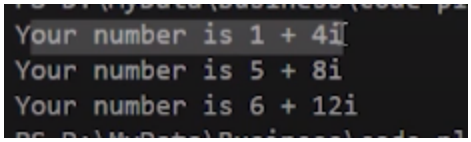


Figure 1: Complex Program Output

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.

```
#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 contains 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.

*/
```

Constructors

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

```
#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";
}
```

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;
}
```

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.
- the 2nd function “printNumber” is called by the objects “c1”, “c2”, and “c3”.

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

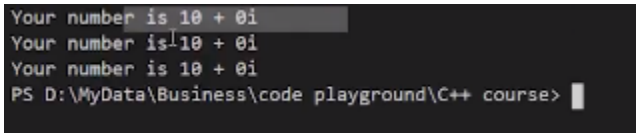


Figure 1: Program Output

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

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";
}
```

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.

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

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

        return 0;
}
```

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.

```
o tut30 } ; if ($?) { .\tut30
Your number is 4 + 6i
Your number is 5 + 7i
PS D:\MyData\Business\code pla
```

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;
        }

};
```

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.

```
#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";
}
```

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.

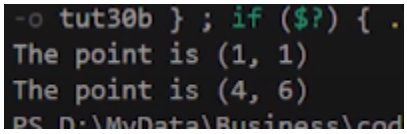


Figure 2: Program Output 2

Constructor Overloading

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;
        }
};
```

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;
}
```

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.

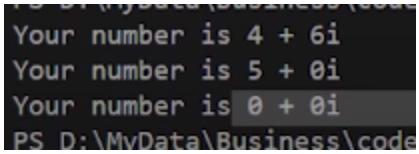


Figure 1: Program Output

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.

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();
};
```

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;
}
```

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;
}
```

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.
- The 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.
- The 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”.
- the 8th function “show” is called which will print the values of data members.

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

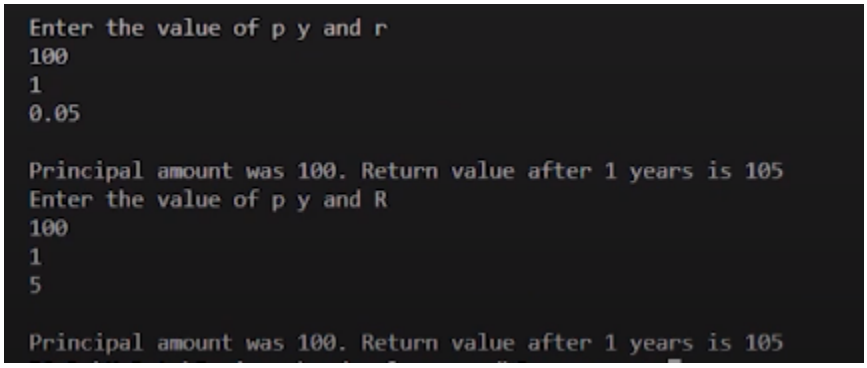


Figure 1: Program Output

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

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, the compiler will supply its own copy constructor. An example program to demonstrate the concept of a Copy constructor in C++ is shown below.

Syntax :-

```
ClassName (const ClassName &old_obj);
```

```
#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;
        }
};
```

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”.

```
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;
}
```

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”

Destructor

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.

```
Syntax:-
~constructor-name();
```

```
#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--;
        }
};
```

Inheritance & Its Different Types

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++

Syntax:-

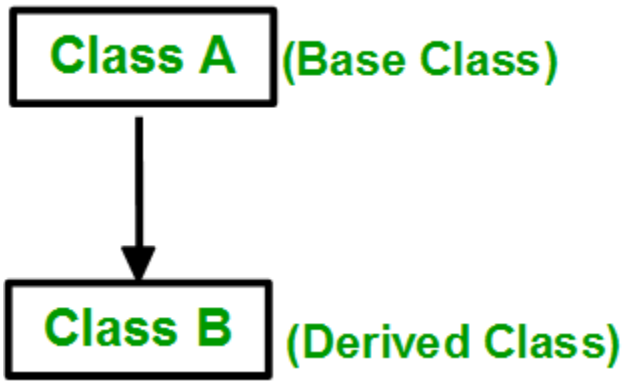
```
class subclass_name : access_mode base_class_name
{
  // body of subclass
};
```

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.




```
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;
}
int main()
{
        Derived der;
        der.setData();
        der.process();
        der.display();

        return 0;
}
```

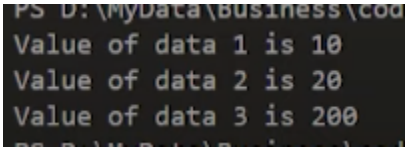
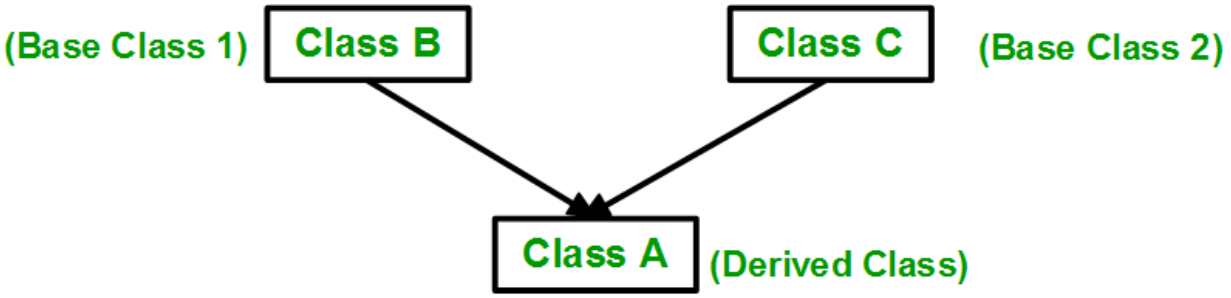


Figure 1: Program Output

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.



Syntax:-

```
class subclass_name : access_mode base_class1, access_mode base_class2, ....
{
  // body of subclass
};
```

```
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;
        }
};
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;
        }
};

int main()
{
        Derived paddy;
        paddy.set_base1int(25);
        paddy.set_base2int(5);
        paddy.set_base3int(15);
        paddy.show();

        return 0;
}
```

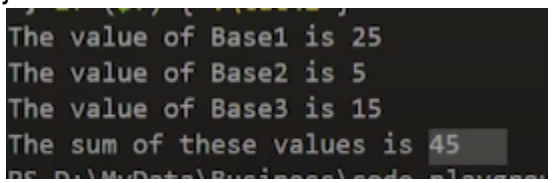
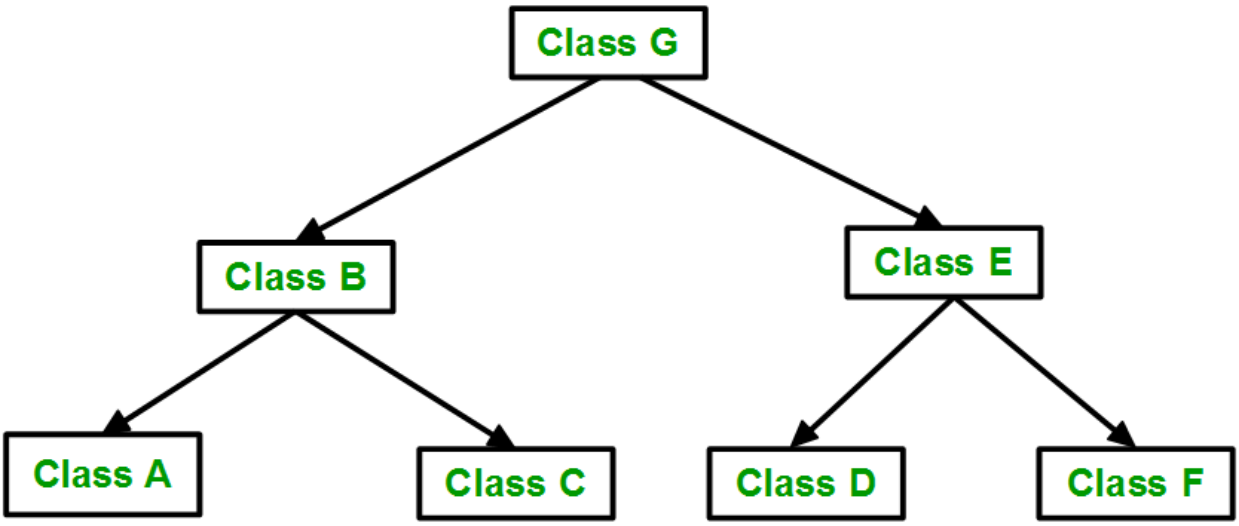


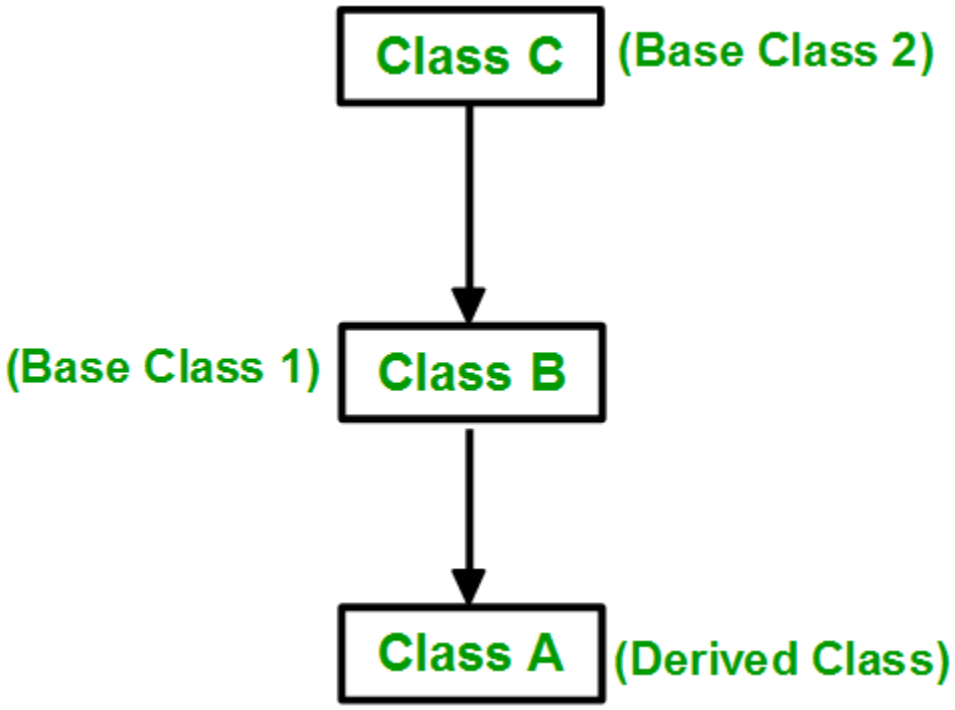
Figure 1: Program Output

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.

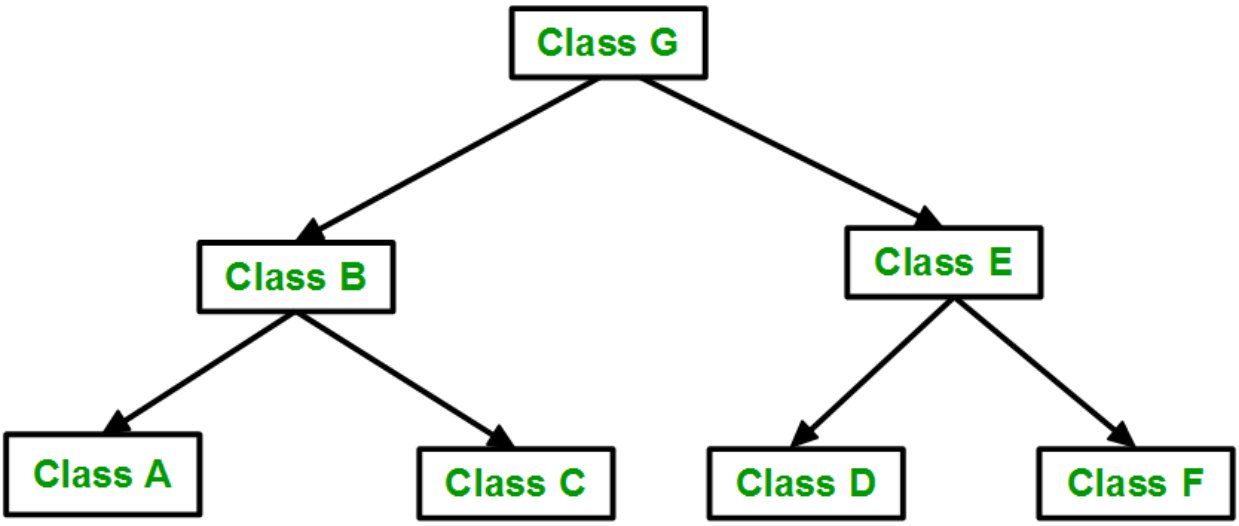
```
#include <iostream>
using namespace std;

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;
}

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;
}

class Result : public Exam
{
        float percentage;

public:
        void display_results()
        {
        get_roll_number();
        get_marks();
        cout << "Your result is " << (maths + physics) / 2 << "%" << endl;
        }
};

int main()
{
        Result paddy;
        paddy.set_roll_number(420);
        paddy.set_marks(94.0, 90.0);
        paddy.display_results();
        return 0;
}
```

```
PS D:\MyData\Business\code playground\C
The roll number is 420
The marks obtained in maths are: 94
The marks obtained in physics are: 90
Your percentage is 92%
PS D:\MyData\Business\code playground\C
```

Figure 1: Program Output

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.

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

Protected Access Modifier

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”.As shown in the table,

	Public Derivation	Private Derivation	Protected Derivation
Private members	Not Inherited	Not Inherited	Not Inherited
Protected members	Protected	Private	Protected
Public members	Public	Private	Protected

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.

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 operations of your choice 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 calculators.
- How is code reusability implemented?

Virtual Base Class

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.

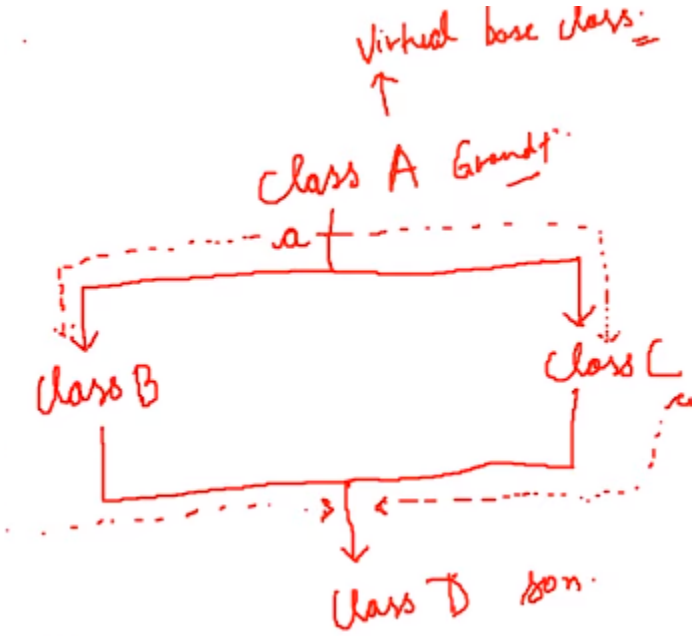


Figure 1: Virtual Base Class Example Diagram

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.

Syntax:-

Syntax 1:

```
class B : virtual public A
{
};
```

Syntax 2:

```
class C : public virtual A
{
};
```

```
#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;
        }
};
int main(){
   Result paddy;
   paddy.set_number(4200);
```

```
    paddy.set_marks(78.9, 99.5);
    paddy.set_score(9);
    paddy.display();
    return 0;
}
```

```
Your roll no is 4200
You result is here:
Maths: 78.9
Physics: 99.5
Your PT score is 9
Your total score is: 187.4
```

Figure 2: Program Output

Constructors in Derived

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

```
Derived-Constructor (arg1, arg2, arg3….): Base 1-Constructor (arg1,arg2), Base
2-Constructor(arg3,arg4)
{
….
} Base 1-Constructor (arg1,arg2)
```

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 inheritance, base classes are constructed in the order in which they appear in the class declaration. For example if there are three classes “A”, “B”, and “C”, and the class “C” inherits 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” inherits classes “A” and the class “C” inherits 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 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

```
/*
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()
};

*/
```

```
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;
        }
};
int main(){
        Derived paddy(1, 2, 3, 4);
        paddy.printDataBase1();
        paddy.printDataBase2();
        paddy.printDataDerived();
        return 0;
}
```

The output of the following program is shown below,

```
Base2 class constructor called
Base1 class constructor called
Derived class constructor called
The value of data1 is 1
The value of data2 is 2
The value of derived1 is 3
The value of derived2 is 4
```

Figure 1: Program Output

Initialization list in Constructors

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;
}
```

```
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;
}
```

The output of the following program is shown below,

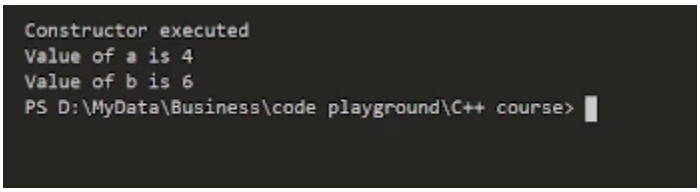


Figure 1: Program Output

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(){
        int a = 4;
        int* ptr = &a;
        cout<<"The value of a is "<<*(ptr)<<endl;

        return 0;
}
```

New Keyword

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

int main(){

        float *p = new float(40.78);
        cout << "The value at address p is " << *(p) << endl;

        return 0;
}
```

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
```

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;
}
```

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
```

Pointers to Objects and Arrow Operator

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 a 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;
}
```

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 objects but still it will give the same result because the pointer is pointing to the address of that object.

The output of the following program is shown below,

```
The real part is 1
The imaginary part is 54
```

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;
}
```

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,

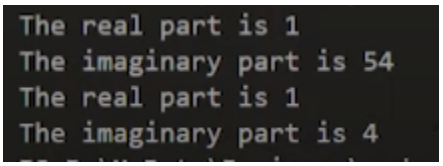


Figure 2: Pointer to Objects Program 2 Output

this Pointer

‘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;
        }
};
int main(){
   A a;
   a.setData(4);
   a.getData();
   return 0;
}
```

The input and output of the following program is shown below,

```
The value of a is 4
```

```
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;
}
```

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

“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
- Runtime 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.

Runtime 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.

Pointer to Derived

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<<"Displaying Base class variable var_base "<<var_base<<endl;
		cout<<"Displaying Derived class variable var_derived "<<var_derived<<endl;
	}
};
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;
}
```

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

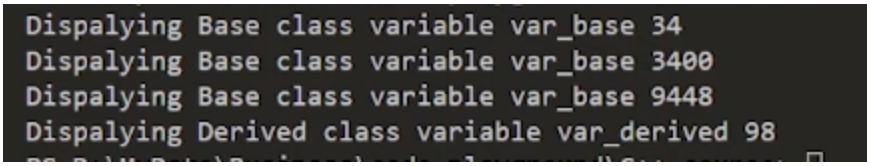


Figure 1: Program Output

Virtual Functions

Virtual Functions in C++

A member function in the base class which is declared using a 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 Displaying Base class variable var_base "<<var_base<<endl;
        }
};

class DerivedClass : public BaseClass{
        public:
        int var_derived=2;
        void display(){
                cout<<"2 Displaying Base class variable var_base "<<var_base<<endl;
                cout<<"2 Displaying Derived class variable var_derived "<<var_derived<<endl;
        }
};
```

Code Snippet 1: Virtual Function Example Program

As shown in code snippet 1,

1. We created a class “BaseClass” which contains the 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;
}
```

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

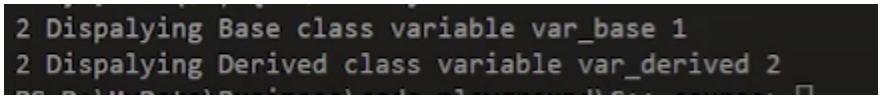


Figure 1: Program Output

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

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 the compiler will throw an error.

To demonstrate the concept of abstract class and pure virtual function an example program is shown below.

```
class YT{
        protected:
        string title;
```

```
        float rating;
        public:
        YT(string s, float r){
        title =  s;
        rating = r;
        }
        virtual void display()=0;
};
```

Code Snippet 1: Code with paddy 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 “YT” 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 YTVideo: public YT
{
        float videoLength;
        public:
        YTVideo(string s, float r, float vl): YT(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;
        }
};
```

Code Snippet 2: Code with paddy Video Class

As shown in code snippet 2,

1. We created a class “CHWVideo” which is inheriting “YT” class and contains private data members “videoLength” which has “float” data type.
2. The class “YTVideo” 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 YTText: public YT
{
        int words;
        public:
        YTText(string s, float r, int wc): YT(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;
        }
};
```

Code Snippet 3: Code with paddy Text Class

As shown in code snippet 3,

1. We created a class “CHWText” which is inheriting “YT” class and contains private data members “words” which has “int” data type.
2. The class “YTText” 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 paddy Video
        title = "Django tutorial";
        vlen = 4.56;
        rating = 4.89;
        YTVideo djVideo(title, rating, vlen);

        // for Code With paddy Text
```

```
        title = "Django tutorial Text";
        words = 433;
        rating = 4.19;
        YTText djText(title, rating, words);

        YT* tuts[2];
        tuts[0] = &djVideo;
        tuts[1] = &djText;

        tuts[0]->display();
        tuts[1]->display();

        return 0;
}
```

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 paddy 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 “YTVideo” and the variables “title”, “rating” and “vlen” are passed to it.
4. For the code with paddy 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 “YTText” and the variables “title”, “rating” and “words” are passed to it.
6. Two pointers array “tuts” is created of the “YT” 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.

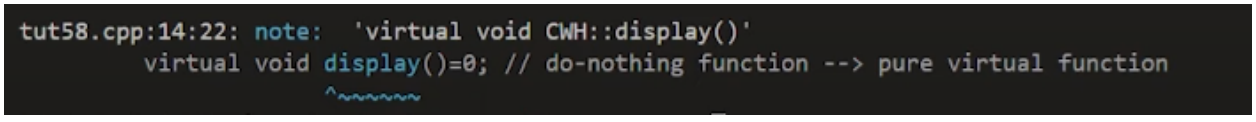


Figure 1: Program Error

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

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.

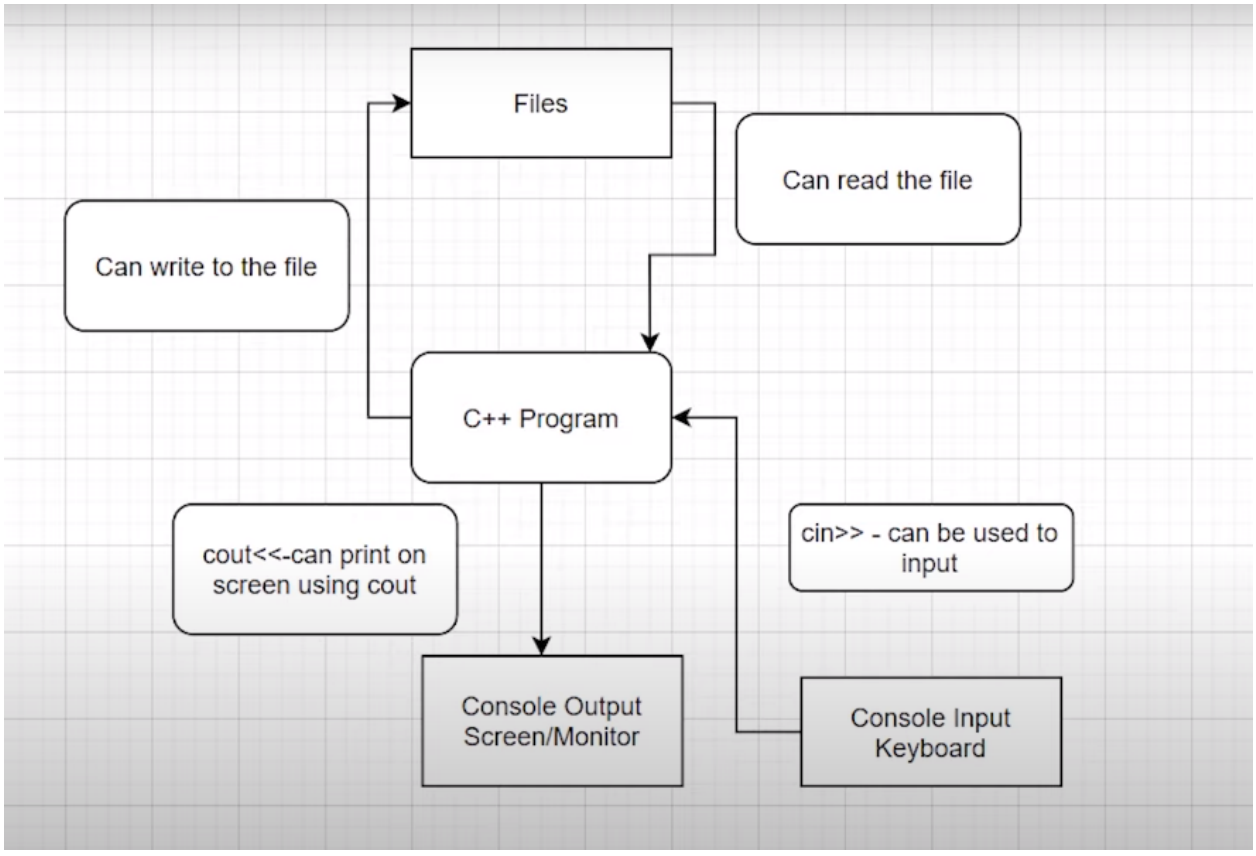


Figure 1: File Read and Write Diagram

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

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 = "paddy bhai";
        // Opening files using constructor and writing it
        ofstream out("sample60.txt"); // Write operation
        out<<st;

        return 0;
}
```

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 “paddy 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”

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;
}
```

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

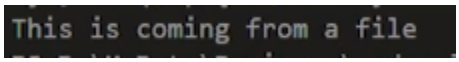


Figure 2: Reading File Operation Output

Closing Files

```
<object_name>.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;
}
```

open() and eof() functions

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;
}
```

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>

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;
}
O/P:-
This is me
This is also m
```

C++ Templates

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 utilize 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:**

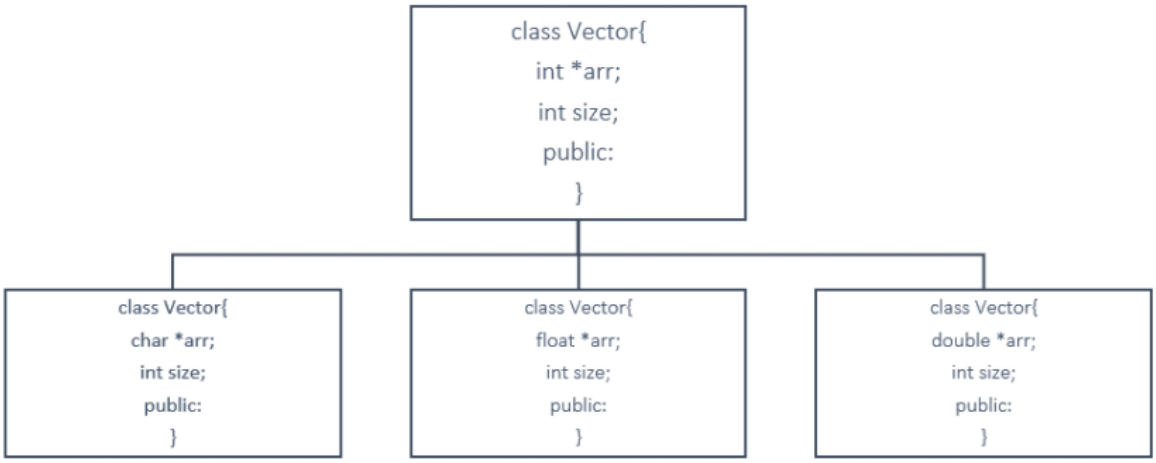
To understand the reason behind using templates, we will have to understand the effort behind declaring classes for different data types. Suppose we want to have a vector for each of the three(can be more) data types, int, float and char. Then we'll obviously write the whole thing again and again making 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:
};
```

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;
}
```

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.

First C++ Template

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>
```

```
#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;
}
```

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>
```

```
#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;
}
```

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 savior.s

And learning it will only benefit you. So why not.

Templates with Multiple Parameters

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
}
```

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;
	}
};
```

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;
        }
};
```

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();
}
```

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>
```

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();
}
```

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>
```

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 realize, can save you a lot of energy and time.

Class Templates with Default Parameters

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 paddy.
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 paddy{
        public:
        T1 a;
        T2 b;
        T3 c;
        paddy(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;
        }
};
```

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 paddy. 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 paddy 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()
{
        paddy<> h(4, 6.4, 'c');
        h.display();
        cout << endl;
        paddy<float, char, char> g(1.6, 'o', 'c');
        g.display();
        return 0;
}
```

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>
```

Function Templates & Function Templates with Parameters

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>
```

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;
}
```

The output of the above program is :

```
The average of these numbers is 3.500000
PS D:\MyData\Business\code playground\C++ course>
```

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;
}
```

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;
}
```

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>
```

```
int main(){
   float a;
   a = funcAverage(5,2.8);
   printf("The average of these numbers is %f",a);
   return 0;
}
```

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>
```

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;
}
```

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.

Member Function Templates & Overloading Template Functions

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 paddy.
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 paddy
{
public:
        T data;
        paddy(T a)
        {
        data = a;
        }
        void display()
        {
        cout << data;
        }
};
```

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 paddy 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 paddy
{
public:
	T data;
	paddy(T a)
	{
	data = a;
	}
	void display();
};

template <class T>
void paddy<T> :: display(){
	cout<<data;
}
```

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()
{
	paddy<int> h(5.7);
	cout << h.data << endl;
	h.display();
	return 0;
}
```

Code Snippet 3: Calling the function from the main

And the output is:

```
5
5
PS D:\MyData\Business\code playground\C++ course>
```

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;
}
```

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;
}
```

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.

Standard Template Library (STL)

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 customize 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.

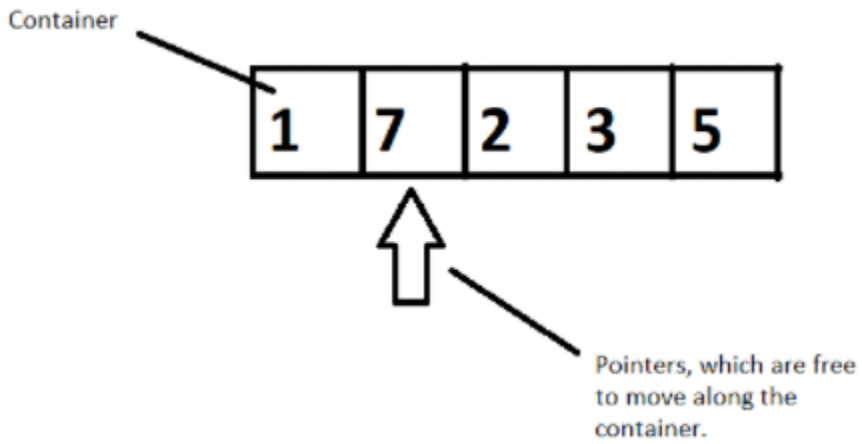


Figure 1: Illustration of how these three components work together

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.