



# SET -1

## C++

### VS-Code



[bonsoninstitutetechology@gmail.com](mailto:bonsoninstitutetechology@gmail.com)

## PROGRAMMING LANGUAGE

# C++ PROGRAMMING

C++ tutorial provides basic and advanced concepts of C++. Our C++ tutorial is designed for beginners and professionals.

C++ is an object-oriented programming language. It is an extension to [C programming](#).

Our C++ tutorial includes all topics of C++ such as first example, control statements, objects and classes, [inheritance](#), [constructor](#), destructor, this, static, polymorphism, abstraction, abstract class, interface, namespace, encapsulation, arrays, strings, exception handling, File IO, etc.

## What is C++

C++ is a general purpose, case-sensitive, free-form programming language that supports object-oriented, procedural and generic programming.

C++ is a middle-level language, as it encapsulates both high and low level language features.

## Object-Oriented Programming (OOPs)

C++ supports the object-oriented programming, the four major pillar of object-oriented programming ([OOPs](#)) used in C++ are:

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

## C++ Standard Libraries

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

- The core library includes the data types, variables and literals, etc.
- The standard library includes the set of functions manipulating strings, files, etc.

- The Standard Template Library (STL) includes the set of methods manipulating a data structure.

## Usage of C++

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

- Window application
- Client-Server application
- Device drivers
- Embedded firmware etc

## C++ Program

In this tutorial, all C++ programs are given with C++ compiler so that you can easily change the C++ program code.

File: main.cpp

```
1. #include <iostream>
2. using namespace std;
3. int main() {
4.     cout << "Hello C++ Programming";
5.     return 0;
6. }
```

## Difference between C and C++

### What is C?

C is a structural or procedural oriented programming language which is machine-independent and extensively used in various applications.

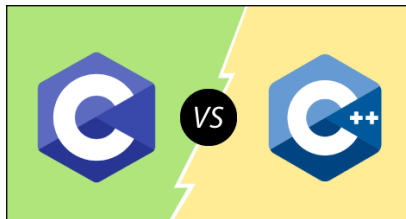
C is the basic programming language that can be used to develop from the operating systems (like Windows) to complex programs like Oracle database, Git, Python interpreter, and many more. C programming language can be called a god's programming language as it forms the base for other programming languages. If we

know the C language, then we can easily learn other programming languages. C language was developed by the great computer scientist Dennis Ritchie at the Bell Laboratories. It contains some additional features that make it unique from other programming languages.

## What is C++?

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

**Let's understand the differences between C and C++.**



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

- **Definition**  
C is a structural programming language, and it does not support classes and objects, while C++ is an object-oriented programming language that supports the concept of classes and objects.
- **Type of programming language**  
C supports the structural programming language where the code is checked line by line, while C++ is an object-oriented programming language that supports the concept of classes and objects.
- **Developer of the language**  
Dennis Ritchie developed C language at Bell Laboratories while Bjarne Stroustrup developed the C++ language at Bell Labs circa 1980.
- **Subset**  
C++ is a superset of C programming language. C++ can run 99% of C code but C language cannot run C++ code.

- **Type of approach**  
C follows the top-down approach, while C++ follows the bottom-up approach. The top-down approach breaks the main modules into tasks; these tasks are broken into sub-tasks, and so on. The bottom-down approach develops the lower level modules first and then the next level modules.
- **Security**  
In C, the data can be easily manipulated by the outsiders as it does not support the encapsulation and information hiding while C++ is a very secure language, i.e., no outsiders can manipulate its data as it supports both encapsulation and data hiding. In C language, functions and data are the free entities, and in C++ language, all the functions and data are encapsulated in the form of objects.
- **Function Overloading**  
Function overloading is a feature that allows you to have more than one function with the same name but varies in the parameters. C does not support the function overloading, while C++ supports the function overloading.
- **Function Overriding**  
Function overriding is a feature that provides the specific implementation to the function, which is already defined in the base class. C does not support the function overriding, while C++ supports the function overriding.
- **Reference variables**  
C does not support the reference variables, while C++ supports the reference variables.
- **Keywords**  
C contains 32 keywords, and C++ supports 52 keywords.
- **Namespace feature**  
A namespace is a feature that groups the entities like classes, objects, and functions under some specific name. C does not contain the namespace feature, while C++ supports the namespace feature that avoids the name collisions.
- **Exception handling**  
C does not provide direct support to the exception handling; it needs to use functions that support exception handling. C++ provides direct support to exception handling by using a try-catch block.

- **Input/Output functions**  
In C, scanf and printf functions are used for input and output operations, respectively, while in C++, cin and cout are used for input and output operations, respectively.
- **Memory allocation and de-allocation**  
C supports calloc() and malloc() functions for the memory allocation, and free() function for the memory de-allocation. C++ supports a new operator for the memory allocation and delete operator for the memory de-allocation.
- **Inheritance**  
Inheritance is a feature that allows the child class to reuse the properties of the parent class. C language does not support the inheritance while C++ supports the inheritance.
- **Header file**  
C program uses **<stdio.h>** header file while C++ program uses **<iostream.h>** header file.

**Let's summarize the above differences in a tabular form.**

No.	C	C++
1)	C follows the <b>procedural style programming</b> .	C++ is multi-paradigm. It supports both <b>procedural and object oriented</b> .
2)	Data is less secured in C.	In C++, you can use modifiers for class members to make it inaccessible for outside users.
3)	C follows the <b>top-down approach</b> .	C++ follows the <b>bottom-up approach</b> .
4)	C does not support function overloading.	C++ supports function overloading.
5)	In C, you can't use functions in structure.	In C++, you can use functions in structure.
6)	C does not support reference variables.	C++ supports reference variables.

7)	In C, <b>scanf()</b> and <b>printf()</b> are mainly used for input/output.	C++ mainly uses stream <b>cin</b> and <b>cout</b> to perform input and output operations.
8)	Operator overloading is not possible in C.	Operator overloading is possible in C++.
9)	C programs are divided into <b>procedures and modules</b>	C++ programs are divided into <b>functions and classes</b> .
10)	C does not provide the feature of namespace.	C++ supports the feature of namespace.
11)	Exception handling is not easy in C. It has to perform using other functions.	C++ provides exception handling using Try and Catch block.
12)	C does not support the inheritance.	C++ supports inheritance.

## C++ Operators

Operators are symbols that perform operations on variables and values. For example, `+` is an operator used for addition, while `-` is an operator used for subtraction.

Operators in C++ can be classified into 6 types:

1. [Arithmetic Operators](#)
2. [Assignment Operators](#)
3. [Relational Operators](#)
4. [Logical Operators](#)
5. [Bitwise Operators](#)
6. [Other Operators](#)

### 1. C++ Arithmetic Operators

Arithmetic operators are used to perform arithmetic operations on variables and data. For example,

```
a + b;
```

Here, the `+` operator is used to add two variables `a` and `b`. Similarly there are various other arithmetic operators in C++.

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

## Example 1: Arithmetic Operators

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

int main() {
    int a, b;
    a = 7;
    b = 2;

    // printing the sum of a and b
    cout << "a + b = " << (a + b) << endl;

    // printing the difference of a and b
    cout << "a - b = " << (a - b) << endl;

    // printing the product of a and b
    cout << "a * b = " << (a * b) << endl;

    // printing the division of a by b
    cout << "a / b = " << (a / b) << endl;

    // printing the modulo of a by b
    cout << "a % b = " << (a % b) << endl;

    return 0;
}
```

[Run Code](#)

## Output

```
a + b = 9
a - b = 5
a * b = 14
a / b = 3
a % b = 1
```



Here, the operators `+`, `-` and `*` compute addition, subtraction, and multiplication respectively as we might have expected.

### / Division Operator

Note the operation `(a / b)` in our program. The `/` operator is the division operator.

As we can see from the above example, if an integer is divided by another integer, we will get the quotient. However, if either divisor or dividend is a floating-point number, we will get the result in decimals.

```
In C++,  
  
7/2 is 3  
  
7.0 / 2 is 3.5  
  
7 / 2.0 is 3.5  
  
7.0 / 2.0 is 3.5
```

## .Increment and Decrement Operators

C++ also provides increment and decrement operators: `++` and `--` respectively.

- `++` increases the value of the operand by 1
- `--` decreases it by 1

For example,

```
int num = 5;  
  
// increment operator  
++num; // 6
```

Here, the code `++num;` increases the value of `num` by 1.

### Example 2: Increment and Decrement Operators

```
// Working of increment and decrement operators  
  
#include <iostream>  
using namespace std;  
  
int main() {  
    int a = 10, b = 100, result_a, result_b;  
  
    // incrementing a by 1 and storing the result in result_a  
    result_a = ++a;  
    cout << "result_a = " << result_a << endl;
```

```

    // decrementing b by 1 and storing the result in result_b
    result_b = --b;
    cout << "result_b = " << result_b << endl;

    return 0;
}

```

[Run Code](#)

## Output

```

result_a = 11
result_b = 99

```

In the above program, we have used the `++` and `--` operators as **prefixes** (`++a` and `--b`). However, we can also use these operators as **postfix** (`a++` and `b--`). To learn more.

## 2. C++ Assignment Operators

In C++, assignment operators are used to assign values to variables. For example,

```

// assign 5 to a
a = 5;

```

Here, we have assigned a value of `5` to the variable `a`.

Operator	Example	Equivalent to
<code>=</code>	<code>a = b;</code>	<code>a = b;</code>
<code>+=</code>	<code>a += b;</code>	<code>a = a + b;</code>
<code>-=</code>	<code>a -= b;</code>	<code>a = a - b;</code>
<code>*=</code>	<code>a *= b;</code>	<code>a = a * b;</code>
<code>/=</code>	<code>a /= b;</code>	<code>a = a / b;</code>
<code>%=</code>	<code>a %= b;</code>	<code>a = a % b;</code>

### Example 3: Assignment Operators

```

#include <iostream>
using namespace std;

```

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

    // 2 is assigned to a
    a = 2;

    // 7 is assigned to b
    b = 7;

    cout << "a = " << a << endl;
    cout << "b = " << b << endl;
    cout << "\nAfter a += b;" << endl;

    // assigning the sum of a and b to a
    a += b; // a = a + b
    cout << "a = " << a << endl;

    return 0;
}
```

[Run Code](#)

## Output

```
a = 2
b = 7

After a += b;
a = 9
```

## 3. C++ Relational Operators

A relational operator is used to check the relationship between two operands. For example,

```
// checks if a is greater than b
a > b;
```

Here, `>` is a relational operator. It checks if `a` is greater than `b` or not.

If the relation is **true**, it returns **1** whereas if the relation is **false**, it returns **0**.

Operator	Meaning	Example
<code>==</code>	Is Equal To	<code>3 == 5</code> gives us <b>false</b>
<code>!=</code>	Not Equal To	<code>3 != 5</code> gives us <b>true</b>
<code>&gt;</code>	Greater Than	<code>3 &gt; 5</code> gives us <b>false</b>

&lt;

Less Than

3 < 5 gives us **true**

&gt;=

Greater Than or Equal To

3 >= 5 give us **false**

&lt;=

Less Than or Equal To

3 <= 5 gives us **true**

## Example 4: Relational Operators

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

int main() {
    int a, b;
    a = 3;
    b = 5;
    bool result;

    result = (a == b);    // false
    cout << "3 == 5 is " << result << endl;

    result = (a != b);    // true
    cout << "3 != 5 is " << result << endl;

    result = a > b;        // false
    cout << "3 > 5 is " << result << endl;

    result = a < b;        // true
    cout << "3 < 5 is " << result << endl;

    result = a >= b;       // false
    cout << "3 >= 5 is " << result << endl;

    result = a <= b;       // true
    cout << "3 <= 5 is " << result << endl;

    return 0;
}
```

[Run Code](#)

### Output

```
3 == 5 is 0
3 != 5 is 1
3 > 5 is 0
3 < 5 is 1
3 >= 5 is 0
3 <= 5 is 1
```

**Note:** Relational operators are used in decision-making and loops.

## 4. C++ Logical Operators

Logical operators are used to check whether an expression is **true** or **false**. If the expression is **true**, it returns **1** whereas if the expression is **false**, it returns **0**.

Operator	Example	Meaning
<code>&amp;&amp;</code>	expression1 <code>&amp;&amp;</code> expression2	Logical AND. True only if all the operands are true.
<code>  </code>	expression1 <code>  </code> expression2	Logical OR. True if at least one of the operands is true.
<code>!</code>	<code>!expression</code>	Logical NOT. True only if the operand is false.

In C++, logical operators are commonly used in decision making. To further understand the logical operators, let's see the following examples,

```
Suppose,
a = 5
b = 8

Then,

(a > 3) && (b > 5) evaluates to true
(a > 3) && (b < 5) evaluates to false

(a > 3) || (b > 5) evaluates to true
(a > 3) || (b < 5) evaluates to true
(a < 3) || (b < 5) evaluates to false

!(a < 3) evaluates to true
!(a > 3) evaluates to false
```

## Example 5: Logical Operators

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

int main() {
    bool result;

    result = (3 != 5) && (3 < 5);    // true
    cout << "(3 != 5) && (3 < 5) is " << result << endl;

    result = (3 == 5) && (3 < 5);    // false
    cout << "(3 == 5) && (3 < 5) is " << result << endl;

    result = (3 == 5) && (3 > 5);    // false
    cout << "(3 == 5) && (3 > 5) is " << result << endl;

    result = (3 != 5) || (3 < 5);    // true
    cout << "(3 != 5) || (3 < 5) is " << result << endl;

    result = (3 != 5) || (3 > 5);    // true
    cout << "(3 != 5) || (3 > 5) is " << result << endl;

    result = (3 == 5) || (3 > 5);    // false
    cout << "(3 == 5) || (3 > 5) is " << result << endl;
```

```

    result = !(5 == 2);    // true
    cout << "!(5 == 2) is " << result << endl;

    result = !(5 == 5);    // false
    cout << "!(5 == 5) is " << result << endl;

    return 0;
}
Run Code

```

### Output

```

(3 != 5) && (3 < 5) is 1
(3 == 5) && (3 < 5) is 0
(3 == 5) && (3 > 5) is 0
(3 != 5) || (3 < 5) is 1
(3 != 5) || (3 > 5) is 1
(3 == 5) || (3 > 5) is 0
!(5 == 2) is 1
!(5 == 5) is 0

```

# C++ Bitwise Operators

In C++, bitwise operators perform operations on integer data at the individual bit-level. These operations include testing, setting, or shifting the actual bits. For example,

```

a & b;
a | b;

```

Here is a list of 6 bitwise operators included in C++.

Operator	Description
&	Bitwise AND Operator
	Bitwise OR Operator
^	Bitwise XOR Operator

`~`

Bitwise Complement Operator

`<<`

Bitwise Shift Left Operator

`>>`

Bitwise Shift Right Operator

These operators are necessary because the Arithmetic-Logic Unit (ALU) present in the computer's CPU carries out arithmetic operations at the bit-level.

**Note:** Bitwise operators can only be used alongside `char` and `int` data types.

## 1. C++ Bitwise AND Operator

The **bitwise AND** `&` operator returns **1** if and only if both the operands are **1**. Otherwise, it returns **0**.

The following table demonstrates the working of the **bitwise AND** operator. Let **a** and **b** be two operands that can only take binary values i.e. **1 and 0**.

a	b	a & b
0	0	0
0	1	0
1	0	0
1	1	1

**Note:** The table above is known as the "Truth Table" for the **bitwise AND** operator.

Let's take a look at the **bitwise AND** operation of two integers 12 and 25:

```
12 = 00001100 (In Binary)
```

25 = 00011001 (In Binary)

//Bitwise AND Operation of 12 and 25

```
00001100
& 00011001
```

---

00001000 = 8 (In decimal)

## Example 1: Bitwise AND

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

int main() {
    // declare variables
    int a = 12, b = 25;

    cout << "a = " << a << endl;
    cout << "b = " << b << endl;
    cout << "a & b = " << (a & b) << endl;

    return 0;
}
```

[Run Code](#)

### Output

```
a = 12
b = 25
a & b = 8
```

In the above example, we have declared two variables `a` and `b`. Here, notice the line,

```
cout << "a & b = " << (a & b) << endl;
```

Here, we are performing **bitwise AND** between variables `a` and `b`.



## 2. C++ Bitwise OR Operator

The **bitwise OR** `|` operator returns **1** if at least one of the operands is **1**.

Otherwise, it returns **0**.

The following truth table demonstrates the working of the **bitwise OR** operator. Let **a** and **b** be two operands that can only take binary values i.e. **1** or **0**.

a	b	a   b
0	0	0
0	1	1
1	0	1
1	1	1

Let us look at the **bitwise OR** operation of two integers **12** and **25**:

```
12 = 00001100 (In Binary)
```

```
25 = 00011001 (In Binary)
```

```
Bitwise OR Operation of 12 and 25
```

```
  00001100
| 00011001
-----
  00011101 = 29 (In decimal)
```

### Example 2: Bitwise OR

```
#include <iostream>

int main() {
    int a = 12, b = 25;
```

```

cout << "a = " << a << endl;
cout << "b = " << b << endl;
cout << "a | b = " << (a | b) << endl;

return 0;
}

```

[Run Code](#)

## Output

```

a = 12
b = 25
a | b = 29

```

The **bitwise OR** of `a = 12` and `b = 25` gives `29`.

## 3. C++ Bitwise XOR Operator

The **bitwise XOR** `^` operator returns **1** if and only if one of the operands is **1**. However, if both the operands are **0**, or if both are **1**, then the result is **0**.

The following truth table demonstrates the working of the **bitwise XOR** operator. Let **a** and **b** be two operands that can only take binary values i.e. **1** or **0**.

a	b	a ^ b
0	0	0
0	1	1
1	0	1
1	1	0

Let us look at the **bitwise XOR** operation of two integers 12 and 25:

```

12 = 00001100 (In Binary)
25 = 00011001 (In Binary)

```

Bitwise XOR Operation of 12 and 25

```
00001100
^ 00011001
-----
00010101 = 21 (In decimal)
```

## Example 3: Bitwise XOR

```
#include <iostream>

int main() {
    int a = 12, b = 25;
    cout << "a = " << a << endl;
    cout << "b = " << b << endl;
    cout << "a ^ b = " << (a ^ b) << endl;

    return 0;
}
```

[Run Code](#)

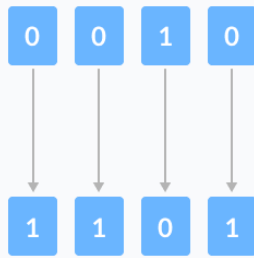
### Output

```
a = 12
b = 25
a ^ b = 21
```

The **bitwise XOR** of `a = 12` and `b = 25` gives `21`.

## 4. C++ Bitwise Complement Operator

The bitwise complement operator is a unary operator (works on only one operand). It is denoted by `~` that changes binary digits **1** to **0** and **0** to **1**.



Bitwise Complement

It is important to note that the **bitwise complement** of any integer **N** is equal to **-(N + 1)**.

For example,

Consider an integer **35**. As per the rule, the bitwise complement of **35** should be **-(35 + 1) = -36**. Now, let's see if we get the correct answer or not.

```
35 = 00100011 (In Binary)
```

```
// Using bitwise complement operator
```

```
~ 00100011
```

```
_____
```

```
11011100
```

In the above example, we get that the bitwise complement of **00100011 (35)** is **11011100**. Here, if we convert the result into decimal we get **220**.

However, it is important to note that we cannot directly convert the result into decimal and get the desired output. This is because the binary result **11011100** is also equivalent to **-36**.

To understand this we first need to calculate the binary output of **-36**. We use 2's complement to calculate the binary of negative integers.

## 2's Complement

The 2's complement of a number **N** gives **-N**.

In binary arithmetic, 1's complement changes **0 to 1** and **1 to 0**.

And, if we add **1** to the result of the 1's complement, we get the 2's complement of the original number.

For example,

```
36 = 00100100 (In Binary)
```

```
1's Complement = 11011011
```

```
2's Complement :
```

```
11011011
```

```
+ 1
```

```
-----
```

```
11011100
```

Here, we can see the 2's complement of **36** (i.e. **-36**) is **11011100**. This value is equivalent to the **bitwise complement of 35** that we have calculated in the previous section.

Hence, we can say that the bitwise complement of 35 = -36.

## Example 4: Bitwise Complement

```
#include <iostream>

int main() {
    int num1 = 35;
    int num2 = -150;
    cout << "~(" << num1 << ") = " << (~num1) << endl;
    cout << "~(" << num2 << ") = " << (~num2) << endl;

    return 0;
}
```

[Run Code](#)

## Output

```
~(35) = -36
~(-150) = 149
```

In the above example, we declared two integer variables `num1` and `num2`, and initialized them with the values of `35` and `-150` respectively.

We then computed their bitwise complement with the codes `(~num1)` and `(~num2)` respectively and displayed them on the screen.

```
The bitwise complement of 35 = - (35 + 1) = -36
```

```
i.e. ~35 = -36
```

```
The bitwise complement of -150 = - (-150 + 1) = - (-149) = 149
```

```
i.e. ~(-150) = 149
```

This is exactly what we got in the output.

## C++ Shift Operators

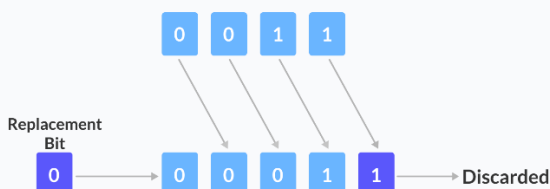
There are two shift operators in C++ programming:

- Right shift operator `>>`
- Left shift operator `<<`

### 5. C++ Right Shift Operator

The **right shift operator** shifts all bits towards the right by a certain number of **specified bits**. It is denoted by `>>`.

When we shift any number to the right, the **least significant bits** are discarded, while the **most significant bits** are replaced by zeroes.

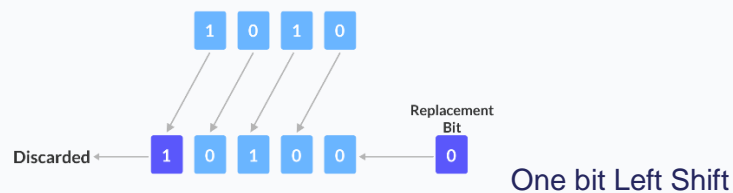


### One bit Right Shift

As we can see from the image above, we have a **4-bit number**. When we perform a **one-bit** right shift operation on it, each individual bit is shifted to the right by 1 bit. As a result, the right-most bit is discarded, while the left-most bit remains vacant. This vacancy is replaced by a **0**.

## 6. C++ Left Shift Operator

The **left shift operator** shifts all bits towards the left by a certain number of **specified bits**. It is denoted by `<<`.



As we can see from the image above, we have a **4-bit number**. When we perform a **1 bit** left shift operation on it, each individual bit is shifted to the left by 1 bit. As a result, the left-most bit is discarded, while the right-most bit remains vacant. This vacancy is replaced by a **0**.

### Example 5: Shift Operators

```
#include <iostream>

int main() {

    // declaring two integer variables
    int num = 212, i;

    // Shift Right Operation
    cout << "Shift Right:" << endl;

    // Using for loop for shifting num right from 0 bit to 3 bits
    for (i = 0; i < 4; i++) {
        cout << "212 >> " << i << " = " << (212 >> i) << endl;
    }
}
```

```
// Shift Left Operation
cout << "\nShift Left:" << endl;

// Using for loop for shifting num left from 0 bit to 3 bits
for (i = 0; i < 4; i++) {
    cout << "212 << " << i << " = " << (212 << i) << endl;
}

return 0;
}
```

[Run Code](#)

## Output

```
Shift Right:
212 >> 0 = 212
212 >> 1 = 106
212 >> 2 = 53
212 >> 3 = 26

Shift Left:
212 << 0 = 212
212 << 1 = 424
212 << 2 = 848
212 << 3 = 1696
```

From the output of the program above, we can infer that, for any number **N**, the results of the shift right operator are:

```
N >> 0 = N

N >> 1 = (N >> 0) / 2

N >> 2 = (N >> 1) / 2

N >> 3 = (N >> 2) / 2
```

and so on.

Similarly, the results of the shift left operator are:



```
N << 0 = N
```

```
N << 1 = (N << 0) * 2
```

```
N << 2 = (N << 1) * 2
```

```
N << 3 = (N << 2) * 2
```

and so on.

Hence we can conclude that,

```
N >> m = [ N >> (m-1) ] / 2
```

```
N << m = [ N << (m-1) ] * 2
```

## Bitwise Shift in Actual Practice

In the above example, note that the `int` data type stores numbers in **32-bits** i.e. an `int` value is represented by **32 binary digits**.

However, our explanation for the bitwise shift operators used numbers represented in **4-bits**.

For example, the base-10 number **13** can be represented in 4-bit and 32-bit as:

```
4-bit Representation of 13 = 1101
```

```
32-bit Representation of 13 = 00000000 00000000 00000000 00001101
```

As a result, the **bitwise left-shift** operation for **13** (and any other number) can be different depending on the number of bits they are represented by.

Because in **32-bit** representation, there are many more bits that can be shifted left when compared to **4-bit** representation.

# C++ if, if...else and Nested if...else

In computer programming, we use the `if...else` statement to run one block of code under certain conditions and another block of code under different conditions.

For example, assigning grades (A, B, C) based on marks obtained by a student.

1. if the percentage is above **90**, assign grade **A**
2. if the percentage is above **75**, assign grade **B**
3. if the percentage is above **65**, assign grade **C**

## C++ if Statement

### Syntax

```
if (condition) {  
    // body of if statement  
}
```

The `if` statement evaluates the `condition` inside the parentheses `( )`.

1. If the `condition` evaluates to `true`, the code inside the body of `if` is executed.
2. If the `condition` evaluates to `false`, the code inside the body of `if` is skipped.

**Note:** The code inside `{ }` is the body of the `if` statement.

### Condition is true

```
int number = 5;

if (number > 0) {
    // code
}

// code after if
```

### Condition is false

```
int number = 5;

if (number < 0) {
    // code
}

// code after if
```

How if Statement Works

## Example 1: C++ if Statement

```
// Program to print positive number entered by the user
// If the user enters a negative number, it is skipped

#include <iostream>
using namespace std;

int main() {

    int number;

    cout << "Enter an integer: ";
    cin >> number;

    // checks if the number is positive
    if (number > 0) {
        cout << "You entered a positive integer: " << number << endl;
    }

    cout << "This statement is always executed.";

    return 0;
}
Run Code
```

### Output 1

```
Enter an integer: 5
You entered a positive number: 5
```

This statement is always executed.

When the user enters `5`, the condition `number > 0` is evaluated to `true` and the statement inside the body of `if` is executed.

## Output 2

```
Enter a number: -5
This statement is always executed.
```

When the user enters `-5`, the condition `number > 0` is evaluated to `false` and the statement inside the body of `if` is not executed.

## C++ if...else

The `if` statement can have an optional `else` clause.

### Syntax

```
if (condition) {
    // block of code if condition is true
}
else {
    // block of code if condition is false
}
```

The `if...else` statement evaluates the `condition` inside the parenthesis.

#### Condition is true

```
int number = 5;

if (number > 0) {
    // code
}
else {
    // code
}

// code after if...else
```

#### Condition is false

```
int number = 5;

if (number < 0) {
    // code
}
else {
    // code
}

// code after if...else
```

### How if...else Statement Works

If the `condition` evaluates `true`,

1. the code inside the body of `if` is executed
2. the code inside the body of `else` is skipped from execution

If the `condition` evaluates `false`,

1. the code inside the body of `else` is executed
2. the code inside the body of `if` is skipped from execution

## Example 2: C++ if...else Statement

```
// Program to check whether an integer is positive or negative
// This program considers 0 as a positive number

#include <iostream>
using namespace std;

int main() {

    int number;

    cout << "Enter an integer: ";
    cin >> number;

    if (number >= 0) {
        cout << "You entered a positive integer: " << number << endl;
    }
    else {
        cout << "You entered a negative integer: " << number << endl;
    }

    cout << "This line is always printed.";

    return 0;
}
Run Code
```

### Output 1

```
Enter an integer: 4
You entered a positive integer: 4.
This line is always printed.
```

In the above program, we have the condition `number >= 0`. If we enter the number greater or equal to `0`, then the condition evaluates `true`.

Here, we enter `4`. So, the condition is `true`. Hence, the statement inside the body of `if` is executed.

## Output 2

```
Enter an integer: -4
You entered a negative integer: -4.
This line is always printed.
```

Here, we enter `-4`. So, the condition is `false`. Hence, the statement inside the body of `else` is executed.

## C++ if...else...else if statement

The `if...else` statement is used to execute a block of code among two alternatives. However, if we need to make a choice between more than two alternatives, we use the `if...else if...else` statement.

### Syntax

```
if (condition1) {
    // code block 1
}
else if (condition2){
    // code block 2
}
else {
    // code block 3
}
```

Here,

1. If `condition1` evaluates to `true`, the `code block 1` is executed.
2. If `condition1` evaluates to `false`, then `condition2` is evaluated.
3. If `condition2` is `true`, the `code block 2` is executed.
4. If `condition2` is `false`, the `code block 3` is executed.

#### 1st Condition is true

```
int number = 2;
if (number > 0) {
    // code
}
else if (number == 0){
    // code
}
else {
    //code
}
//code after if
```

#### 2nd Condition is true

```
int number = 0;
if (number > 0) {
    // code
}
else if (number == 0){
    // code
}
else {
    //code
}
//code after if
```

#### All Conditions are false

```
int number = -2;
if (number > 0) {
    // code
}
else if (number == 0){
    // code
}
else {
    //code
}
//code after if
```

How

if...else if...else Statement Works

**Note:** There can be more than one `else if` statement but only one `if` and `else` statements.

## Example 3: C++ if...else...else if

```
// Program to check whether an integer is positive, negative or zero

#include <iostream>
using namespace std;

int main() {

    int number;

    cout << "Enter an integer: ";
    cin >> number;

    if (number > 0) {
        cout << "You entered a positive integer: " << number << endl;
    }
    else if (number < 0) {
        cout << "You entered a negative integer: " << number << endl;
    }
}
```

```
else {  
    cout << "You entered 0." << endl;  
}  
  
cout << "This line is always printed."  
  
return 0;  
}  
Run Code
```

## Output 1

```
Enter an integer: 1  
You entered a positive integer: 1.  
This line is always printed.
```

## Output 2

```
Enter an integer: -2  
You entered a negative integer: -2.  
This line is always printed.
```

## Output 3

```
Enter an integer: 0  
You entered 0.  
This line is always printed.
```

In this program, we take a number from the user. We then use the `if...else if...else` ladder to check whether the number is positive, negative, or zero.

If the number is greater than `0`, the code inside the `if` block is executed. If the number is less than `0`, the code inside the `else if` block is executed.

Otherwise, the code inside the `else` block is executed.

## C++ Nested if...else

Sometimes, we need to use an `if` statement inside another `if` statement. This is known as nested `if` statement.



Think of it as multiple layers of `if` statements. There is a first, outer `if` statement, and inside it is another, inner `if` statement.

## Syntax

```
// outer if statement
if (condition1) {

    // statements

    // inner if statement
    if (condition2) {
        // statements
    }
}
```

## Notes:

1. We can add `else` and `else if` statements to the inner `if` statement as required.
2. The inner `if` statement can also be inserted inside the outer `else` or `else if` statements (if they exist).
3. We can nest multiple layers of `if` statements.

## Example 4: C++ Nested if

```
// C++ program to find if an integer is positive, negative or zero
// using nested if statements

#include <iostream>
using namespace std;

int main() {

    int num;

    cout << "Enter an integer: ";
    cin >> num;

    // outer if condition
```

```
if (num != 0) {  
  
    // inner if condition  
    if (num > 0) {  
        cout << "The number is positive." << endl;  
    }  
    // inner else condition  
    else {  
        cout << "The number is negative." << endl;  
    }  
}  
// outer else condition  
else {  
    cout << "The number is 0 and it is neither positive nor negative." << endl;  
}  
  
cout << "This line is always printed." << endl;  
  
return 0;  
}
```

[Run Code](#)

### Output 1

```
Enter an integer: 35  
The number is positive.  
This line is always printed.
```

### Output 2

```
Enter an integer: -35  
The number is negative.  
This line is always printed.
```

### Output 3

```
Enter an integer: 0  
The number is 0 and it is neither positive nor negative.  
This line is always printed.
```

In the above example,

1. We take an integer as an input from the user and store it in the variable `num`.
2. We then use an `if...else` statement to check whether `num` is not equal to `0`.
  - a. If `true`, then the **inner** `if...else` statement is executed.
  - b. If `false`, the code inside the **outer** `else` condition is executed, which prints `"The number is 0 and it is neither positive nor negative."`
3. The **inner** `if...else` statement checks whether the input number is positive i.e. if `num` is greater than `0`.
  - a. If `true`, then we print a statement saying that the number is positive.
  - b. If `false`, we print that the number is negative.

**Note:** As you can see, nested `if...else` makes your logic complicated. If possible, you should always try to avoid nested `if...else`.

## Body of if...else With Only One Statement

If the body of `if...else` has only one statement, you can omit `{ }` in the program. For example, you can replace

```
int number = 5;

if (number > 0) {
    cout << "The number is positive." << endl;
}
else {
    cout << "The number is negative." << endl;
}
```

with

```
int number = 5;

if (number > 0)
    cout << "The number is positive." << endl;
else
```

```
cout << "The number is negative." << endl;
```

The output of both programs will be the same.

**Note:** Although it's not necessary to use `{ }` if the body of `if...else` has only one statement, using `{ }` makes your code more readable.

## More on Decision Making

The **ternary operator** is a concise, inline method used to execute one of two expressions based on a condition. To learn more, visit [C++ Ternary Operator](#). If we need to make a choice between more than one alternatives based on a given test condition, the `switch` statement can be used. To learn more, visit [C++ switch](#).

### Also Read:

1. [C++ Program to Check Whether Number is Even or Odd](#)
2. [C++ Program to Check Whether a Character is Vowel or Consonant.](#)
3. [C++ Program to Find Largest Number Among Three Numbers](#)

## C++ while and do...while Loop

In computer programming, loops are used to repeat a block of code.

For example, let's say we want to show a message 100 times. Then instead of writing the print statement 100 times, we can use a loop.

That was just a simple example; we can achieve much more efficiency and sophistication in our programs by making effective use of loops.

There are **3** types of loops in C++.

1. `for` loop
2. `while` loop
3. `do...while` loop

In the previous tutorial, we learned about the [C++ for loop](#). Here, we are going to learn about `while` and `do...while` loops.

## C++ while Loop

The syntax of the `while` loop is:

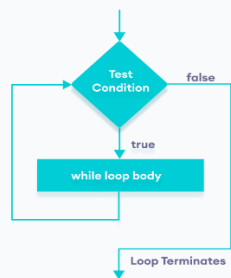
```
while (condition) {  
    // body of the loop  
}
```

Here,

- A `while` loop evaluates the `condition`
- If the `condition` evaluates to `true`, the code inside the `while` loop is executed.
- The `condition` is evaluated again.
- This process continues until the `condition` is `false`.
- When the `condition` evaluates to `false`, the loop terminates.

To learn more about the `conditions`, visit [C++ Relational and Logical Operators](#).

## Flowchart of while Loop



Flowchart of C++ while loop

## Example 1: Display Numbers from 1 to 5

```
// C++ Program to print numbers from 1 to 5
```

```
#include <iostream>
```

```
using namespace std;
```

```
int main() {
```

```
    int i = 1;
```

```
    // while loop from 1 to 5
```

```
    while (i <= 5) {
```

```
        cout << i << " ";
```

```
        ++i;
```

```
    }
```

```
    return 0;
```

```
}
```

[Run Code](#)

## Output

```
1 2 3 4 5
```

Here is how the program works.

Iteration	Variable	$i \leq 5$	Action
1st	<code>i = 1</code>	true	1 is printed and <code>i</code> is increased to 2.
2nd	<code>i = 2</code>	true	2 is printed and <code>i</code> is increased to 3.
3rd	<code>i = 3</code>	true	3 is printed and <code>i</code> is increased to 4
4th	<code>i = 4</code>	true	4 is printed and <code>i</code> is increased to 5.
5th	<code>i = 5</code>	true	5 is printed and <code>i</code> is increased to 6.
6th	<code>i = 6</code>	false	The loop is terminated

## Example 2: Sum of Positive Numbers Only

```
// program to find the sum of positive numbers
// if the user enters a negative number, the loop ends
// the negative number entered is not added to the sum

#include <iostream>
using namespace std;

int main() {
    int number;
    int sum = 0;

    // take input from the user
    cout << "Enter a number: ";
    cin >> number;

    while (number >= 0) {
        // add all positive numbers
        sum += number;

        // take input again if the number is positive
        cout << "Enter a number: ";
        cin >> number;
    }

    // display the sum
    cout << "\nThe sum is " << sum << endl;

    return 0;
}
```

[Run Code](#)

## Output

```
Enter a number: 6
Enter a number: 12
Enter a number: 7
Enter a number: 0
Enter a number: -2
```

```
The sum is 25
```

In this program, the user is prompted to enter a number, which is stored in the variable `number`.

In order to store the sum of the numbers, we declare a variable `sum` and initialize it to the value of `0`.

The `while` loop continues until the user enters a negative number. During each iteration, the number entered by the user is added to the `sum` variable.

When the user enters a negative number, the loop terminates. Finally, the total sum is displayed.

## C++ do...while Loop

The `do...while` loop is a variant of the `while` loop with one important difference: the body of `do...while` loop is executed once before the `condition` is checked.

Its syntax is:

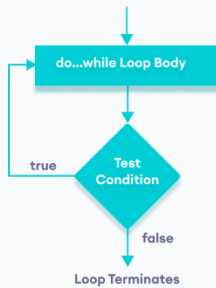
```
do {  
    // body of loop;  
}  
while (condition);
```

Here,

- The body of the loop is executed at first. Then the `condition` is evaluated.
- If the `condition` evaluates to `true`, the body of the loop inside the `do` statement is executed again.
- The `condition` is evaluated once again.
- If the `condition` evaluates to `true`, the body of the loop inside the `do` statement is executed again.
- This process continues until the `condition` evaluates to `false`. Then the loop stops.

## Flowchart of do...while Loop





Flowchart of C++ do...while loop

## Example 3: Display Numbers from 1 to 5

// C++ Program to print numbers from 1 to 5

```
#include <iostream>
```

```
using namespace std;
```

```
int main() {
    int i = 1;

    // do...while loop from 1 to 5
    do {
        cout << i << " ";
        ++i;
    }
    while (i <= 5);

    return 0;
}
```

[Run Code](#)

## Output

```
1 2 3 4 5
```

Here is how the program works.

Iteration	Variable	$i \leq 5$	Action
	<code>i = 1</code>	not checked	<code>1</code> is printed and <code>i</code> is increased to 2
1st	<code>i = 2</code>	<code>true</code>	<code>2</code> is printed and <code>i</code> is increased to 3

2nd	<code>i = 3</code>	<code>true</code>	3 is printed and <code>i</code> is increased to 4
3rd	<code>i = 4</code>	<code>true</code>	4 is printed and <code>i</code> is increased to 5
4th	<code>i = 5</code>	<code>true</code>	5 is printed and <code>i</code> is increased to 6
5th	<code>i = 6</code>	<code>false</code>	The loop is terminated

## Example 4: Sum of Positive Numbers Only

```
// program to find the sum of positive numbers
// If the user enters a negative number, the loop ends
// the negative number entered is not added to the sum

#include <iostream>
using namespace std;

int main() {
    int number = 0;
    int sum = 0;

    do {
        sum += number;

        // take input from the user
        cout << "Enter a number: ";
        cin >> number;
    }
    while (number >= 0);

    // display the sum
    cout << "\nThe sum is " << sum << endl;

    return 0;
}
```

[Run Code](#)

## Output 1

```
Enter a number: 6
Enter a number: 12
Enter a number: 7
```

```
Enter a number: 0
Enter a number: -2
```

```
The sum is 25
```

Here, the `do...while` loop continues until the user enters a negative number. When the number is negative, the loop terminates; the negative number is not added to the `sum` variable.

## Output 2

```
Enter a number: -6
The sum is 0.
```

The body of the `do...while` loop runs only once if the user enters a negative number.

## Infinite while loop

If the `condition` of a loop is always `true`, the loop runs for infinite times (until the memory is full). For example,

```
// infinite while loop
while(true) {
    // body of the loop
}
```

Here is an example of an infinite `do...while` loop.

```
// infinite do...while loop

int count = 1;

do {
    // body of loop
}
while(count == 1);
```

In the above programs, the `condition` is always `true`. Hence, the loop body will run for infinite times.

## for vs while loops

A `for` loop is usually used when the number of iterations is known. For example,

```
// This loop is iterated 5 times
for (int i = 1; i <=5; ++i) {
    // body of the loop
}
```

Here, we know that the for-loop will be executed 5 times.

However, `while` and `do...while` loops are usually used when the number of iterations is unknown. For example,

```
while (condition) {
    // body of the loop
}
```

- [C++ Program to Display Fibonacci Series](#)
- [C++ Program to Find GCD](#)
- [C++ Program to Find LCM](#)

## C++ break Statement

In C++, the `break` statement terminates the loop when it is encountered.

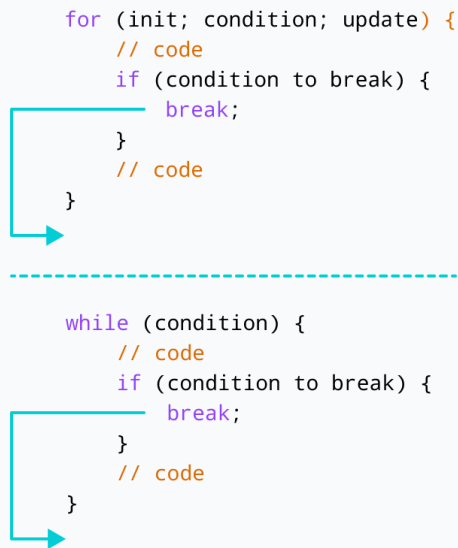
The syntax of the `break` statement is:

```
break;
```

Before you learn about the `break` statement, make sure you know about:

- [C++ for loop](#)
- [C++ if...else](#)
- [C++ while loop](#)

## Working of C++ break Statement



Working of break statement in C++

## Example 1: break with for loop

```
// program to print the value of i

#include <iostream>
using namespace std;

int main() {
    for (int i = 1; i <= 5; i++) {
        // break condition
        if (i == 3) {
            break;
        }
        cout << i << endl;
    }

    return 0;
```

```
}  
Run Code
```

## Output

```
1  
2
```

In the above program, the `for` loop is used to print the value of `i` in each iteration. Here, notice the code:

```
if (i == 3) {  
    break;  
}
```

This means, when `i` is equal to `3`, the `break` statement terminates the loop. Hence, the output doesn't include values greater than or equal to `3`.

Note: The `break` statement is usually used with decision-making statements.

## Example 2: break with while loop

```
// program to find the sum of positive numbers  
// if the user enters a negative numbers, break ends the loop  
// the negative number entered is not added to sum  
  
#include <iostream>  
using namespace std;  
  
int main() {  
    int number;  
    int sum = 0;  
  
    while (true) {  
        // take input from the user  
        cout << "Enter a number: ";  
        cin >> number;  
  
        // break condition  
        if (number < 0) {
```

```

        break;
    }

    // add all positive numbers
    sum += number;
}

// display the sum
cout << "The sum is " << sum << endl;

return 0;
}

```

[Run Code](#)

## Output

```

Enter a number: 1
Enter a number: 2
Enter a number: 3
Enter a number: -5
The sum is 6.

```

In the above program, the user enters a number. The `while` loop is used to print the total sum of numbers entered by the user. Here, notice the code,

```

if(number < 0) {
    break;
}

```

This means, when the user enters a negative number, the `break` statement terminates the loop and codes outside the loop are executed.

The `while` loop continues until the user enters a negative number.

## break with Nested loop

When `break` is used with nested loops, `break` terminates the inner loop. For example,

```

// using break statement inside
// nested for loop

```

```

#include <iostream>
using namespace std;

int main() {
    int number;
    int sum = 0;
    // nested for loops
    // first loop
    for (int i = 1; i <= 3; i++) {
        // second loop
        for (int j = 1; j <= 3; j++) {
            if (i == 2) {
                break;
            }
            cout << "i = " << i << ", j = " << j << endl;
        }
    }
    return 0;
}

```

[Run Code](#)

## Output

```

i = 1, j = 1
i = 1, j = 2
i = 1, j = 3
i = 3, j = 1
i = 3, j = 2
i = 3, j = 3

```

In the above program, the `break` statement is executed when `i == 2`. It terminates the inner loop, and the control flow of the program moves to the outer loop.

Hence, the value of `i = 2` is never displayed in the output.

The `break` statement is also used with the `switch` statement.

# C++ continue Statement



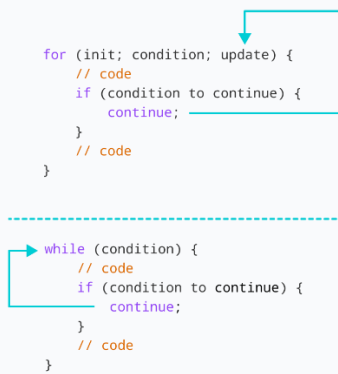
In computer programming, the `continue` statement is used to skip the current iteration of the loop and the control of the program goes to the next iteration. The syntax of the `continue` statement is:

```
continue;
```

Before you learn about the continue statement, make sure you know about,

- [C++ for loop](#)
- [C++ if...else](#)
- [C++ while loop](#)

## Working of C++ continue Statement



```
for (init; condition; update) {  
    // code  
    if (condition to continue) {  
        continue;  
    }  
    // code  
}
```

---

```
while (condition) {  
    // code  
    if (condition to continue) {  
        continue;  
    }  
    // code  
}
```

Working of continue statement in C++

## Example 1: continue with for loop

In a `for` loop, `continue` skips the current iteration and the control flow jumps to the `update` expression.

```
// program to print the value of i
```

```
#include <iostream>
```

```
using namespace std;

int main() {
    for (int i = 1; i <= 5; i++) {
        // condition to continue
        if (i == 3) {
            continue;
        }
        cout << i << endl;
    }

    return 0;
}
Run Code
```

## Output

```
1
2
4
5
```

In the above program, we have used the `for` loop to print the value of `i` in each iteration. Here, notice the code,

```
if (i == 3) {
    continue;
}
```

This means

- When `i` is equal to `3`, the `continue` statement skips the current iteration and starts the next iteration
- Then, `i` becomes `4`, and the `condition` is evaluated again.
- Hence, `4` and `5` are printed in the next two iterations.

**Note:** The `continue` statement is almost always used with decision-making statements.

## Example 2: continue with while loop

In a `while` loop, `continue` skips the current iteration and control flow of the program jumps back to the `while` condition.

```
// program to calculate positive numbers till 50 only
// if the user enters a negative number,
// that number is skipped from the calculation

// negative number -> loop terminate
// numbers above 50 -> skip iteration

#include <iostream>
using namespace std;

int main() {
    int sum = 0;
    int number = 0;

    while (number >= 0) {
        // add all positive numbers
        sum += number;

        // take input from the user
        cout << "Enter a number: ";
        cin >> number;

        // continue condition
        if (number > 50) {
            cout << "The number is greater than 50 and won't be calculated." <<
endl;
            number = 0; // the value of number is made 0 again
            continue;
        }
    }

    // display the sum
    cout << "The sum is " << sum << endl;

    return 0;
}
```

[Run Code](#)

## Output

```
Enter a number: 12
Enter a number: 0
Enter a number: 2
Enter a number: 30
Enter a number: 50
Enter a number: 56
The number is greater than 50 and won't be calculated.
Enter a number: 5
Enter a number: -3
The sum is 99
```

In the above program, the user enters a number. The `while` loop is used to print the total sum of positive numbers entered by the user, as long as the numbers entered are not greater than `50`.

Notice the use of the `continue` statement.

```
if (number > 50){
    continue;
}
```

- When the user enters a number greater than `50`, the `continue` statement skips the current iteration. Then the control flow of the program goes to the `condition` of `while` loop.
- When the user enters a number less than `0`, the loop terminates.

**Note:** The `continue` statement works in the same way for the `do...while` loops.

## continue with Nested loop

When `continue` is used with nested loops, it skips the current iteration of the inner loop. For example,

```
// using continue statement inside
// nested for loop

#include <iostream>
```

```
using namespace std;

int main() {
    int number;
    int sum = 0;

    // nested for loops

    // first loop
    for (int i = 1; i <= 3; i++) {
        // second loop
        for (int j = 1; j <= 3; j++) {
            if (j == 2) {
                continue;
            }
            cout << "i = " << i << ", j = " << j << endl;
        }
    }

    return 0;
}
```

[Run Code](#)

## Output

```
i = 1, j = 1
i = 1, j = 3
i = 2, j = 1
i = 2, j = 3
i = 3, j = 1
i = 3, j = 3
```

In the above program, when the `continue` statement executes, it skips the current iteration in the inner loop. And the control of the program mo

# C++ goto Statement

In C++ programming, the goto statement is used for altering the normal sequence of program execution by transferring control to some other part of the program.

## Syntax of goto Statement

```
goto label;  
... ..  
... ..  
... ..  
label:  
statement;  
... ..
```

In the syntax above, `label` is an **identifier**. When `goto label;` is encountered, the control of program jumps to `label:` and executes the code below it.



Working of goto in C++

## Example: goto Statement

```
// This program calculates the average of numbers entered by the user.  
// If the user enters a negative number, it ignores the number and  
// calculates the average number entered before it.  
  
# include <iostream>  
using namespace std;  
  
int main()  
{  
    float num, average, sum = 0.0;  
    int i, n;
```

```

cout << "Maximum number of inputs: ";
cin >> n;

for(i = 1; i <= n; ++i)
{
    cout << "Enter n" << i << ": ";
    cin >> num;

    if(num < 0.0)
    {
        // Control of the program move to jump:
        goto jump;
    }
    sum += num;
}

jump:
average = sum / (i - 1);
cout << "\nAverage = " << average;
return 0;
}

```

## Output

```

Maximum number of inputs: 10
Enter n1: 2.3
Enter n2: 5.6
Enter n3: -5.6

Average = 3.95

```

# C++ switch...case Statement

The `switch` statement allows us to execute a block of code among many alternatives.

You can do the same thing with the [if...else](#) statement. However, the syntax of the switch statement is much easier to read and write.

## Syntax

```
switch (expression) {  
    case constant1:  
        // code to be executed if  
        // expression is equal to constant1;  
        break;  
  
    case constant2:  
        // code to be executed if  
        // expression is equal to constant2;  
        break;  
    .  
    .  
    .  
    default:  
        // code to be executed if  
        // expression doesn't match any constant  
}
```

## How does the switch statement work?

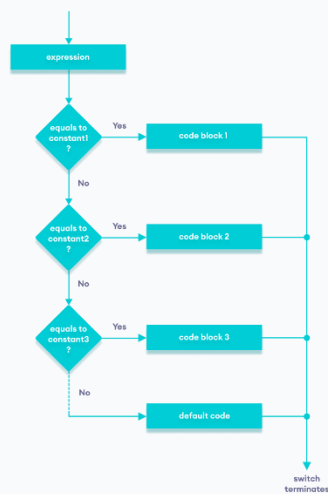
The `expression` is evaluated once and compared with the values of each `case` label.

- If there is a match, the corresponding code after the matching label is executed. For example, if the value of the [variable](#) is equal to `constant2`, the code after `case constant2:` is executed until the [break statement](#) is encountered.
- If there is no match, the code after `default:` is executed.

**Note:** We can do the same thing with the if...else..if ladder. However, the syntax of the switch statement is cleaner and much easier to read and write.

## Flowchart of switch Statement





Flowchart of C++ switch...case statement

## Example: Create a Calculator using the switch Statement

```
// Program to build a simple calculator using switch Statement
#include <iostream>
using namespace std;

int main() {
    char oper;
    float num1, num2;
    cout << "Enter an operator (+, -, *, /): ";
    cin >> oper;
    cout << "Enter two numbers: " << endl;
    cin >> num1 >> num2;

    switch (oper) {
        case '+':
            cout << num1 << " + " << num2 << " = " << num1 + num2;
            break;
        case '-':
            cout << num1 << " - " << num2 << " = " << num1 - num2;
            break;
        case '*':
            cout << num1 << " * " << num2 << " = " << num1 * num2;
            break;
```

```

        case '/':
            cout << num1 << " / " << num2 << " = " << num1 / num2;
            break;
        default:
            // operator is doesn't match any case constant (+, -, *, /)
            cout << "Error! The operator is not correct";
            break;
    }

    return 0;
}

```

[Run Code](#)

## Output 1

```

Enter an operator (+, -, *, /): +
Enter two numbers:
2.3
4.5
2.3 + 4.5 = 6.8

```

## Output 2

```

Enter an operator (+, -, *, /): -
Enter two numbers:
2.3
4.5
2.3 - 4.5 = -2.2

```

## Output 3

```

Enter an operator (+, -, *, /): *
Enter two numbers:
2.3
4.5
2.3 * 4.5 = 10.35

```

## Output 4

```

Enter an operator (+, -, *, /): /
Enter two numbers:
2.3

```

```
4.5
2.3 / 4.5 = 0.511111
```

## Output 5

```
Enter an operator (+, -, *, /): ?
Enter two numbers:
2.3
4.5
Error! The operator is not correct.
```

In the above program, we are using the `switch...case` statement to perform addition, subtraction, multiplication, and division.

### How This Program Works

1. We first prompt the user to enter the desired [operator](#). This input is then stored in the `char` variable named `open`.
2. We then prompt the user to enter two numbers, which are stored in the float variables `num1` and `num2`.
3. The `switch` statement is then used to check the operator entered by the user:
  - a. If the user enters `+`, addition is performed on the numbers.
  - b. If the user enters `-`, subtraction is performed on the numbers.
  - c. If the user enters `*`, multiplication is performed on the numbers.
  - d. If the user enters `/`, division is performed on the numbers.
  - e. If the user enters any other character, the default code is printed.

Notice that the [break statement](#) is used inside each `case` block. This terminates the `switch` statement.

# C++ Functions

A function is a block of code that performs a specific task.

Suppose we need to create a program to create a circle and color it. We can create two functions to solve this problem:

- a function to draw the circle
- a function to color the circle

Dividing a complex problem into smaller chunks makes our program easy to understand and reusable.

There are two types of function:

1. **Standard Library Functions:** Predefined in C++
2. **User-defined Function:** Created by users

In this tutorial, we will focus mostly on user-defined functions.

## C++ User-defined Function

C++ allows the programmer to define their own function.

A user-defined function groups code to perform a specific task and that group of code is given a name (identifier).

When the function is invoked from any part of the program, it all executes the codes defined in the body of the function.

## C++ Function Declaration

The syntax to declare a function is:

```
returnType functionName (parameter1, parameter2,...) {  
    // function body  
}
```

Here's an example of a function declaration.

```
// function declaration
void greet() {
    cout << "Hello World";
}
```

Here,

- the name of the function is `greet()`
- the return type of the function is `void`
- the empty parentheses mean it doesn't have any parameters
- the function body is written inside `{}`

**Note:** We will learn about `returnType` and `parameters` later in this tutorial.

## Calling a Function

In the above program, we have declared a function named `greet()`. To use the `greet()` function, we need to call it.

Here's how we can call the above `greet()` function.

```
int main() {

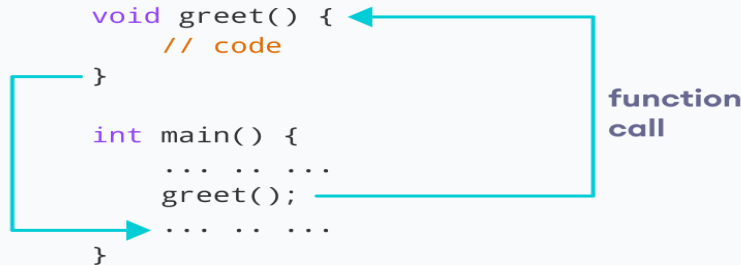
    // calling a function
    greet();

}
```

```
#include<iostream>

void greet() {
    // code
}

int main() {
    ... ..
    greet();
    ... ..
}
```



function call

How Function works in C++

## Example 1: Display a Text

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

// declaring a function
void greet() {
    cout << "Hello there!";
}

int main() {

    // calling the function
    greet();

    return 0;
}
```

[Run Code](#)

### Output

Hello there!

## Function Parameters

As mentioned above, a function can be declared with parameters (arguments). A parameter is a value that is passed when declaring a function.

For example, let us consider the function below:

```
void printNum(int num) {
    cout << num;
}
```

Here, the `int` variable `num` is the function parameter.

We pass a value to the function parameter while calling the function.

```
int main() {
    int n = 7;
```

```
    // calling the function
    // n is passed to the function as argument
    printNum(n);

    return 0;
}
```

## Example 2: Function with Parameters

```
// program to print a text

#include <iostream>
using namespace std;

// display a number
void displayNum(int n1, float n2) {
    cout << "The int number is " << n1;
    cout << "The double number is " << n2;
}

int main() {

    int num1 = 5;
    double num2 = 5.5;

    // calling the function
    displayNum(num1, num2);

    return 0;
}
```

[Run Code](#)

## Output

```
The int number is 5
The double number is 5.5
```

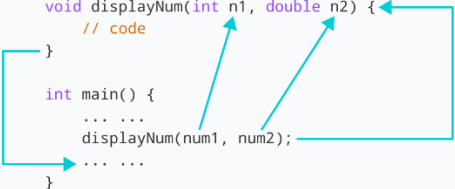
In the above program, we have used a function that has one `int` parameter and one `double` parameter.

We then pass `num1` and `num2` as arguments. These values are stored by the function parameters `n1` and `n2` respectively.

```
#include<iostream>

void displayNum(int n1, double n2) {
    // code
}

int main() {
    ...
    displayNum(num1, num2);
    ...
}
```



The diagram illustrates a function call. A teal arrow originates from the `displayNum(num1, num2);` line in the `main()` function and points to the opening curly brace of the `displayNum` function definition. Another teal arrow points from the `num1` argument to the `int n1` parameter, and a third teal arrow points from the `num2` argument to the `double n2` parameter. A bracket on the right side of the diagram, labeled "function call", spans the distance from the `main()` function to the `displayNum` function definition.

C++ function with parameters

**Note:** The type of the arguments passed while calling the function must match with the corresponding parameters defined in the function declaration.

## Return Statement

In the above programs, we have used `void` in the function declaration. For example,

```
void displayNumber() {
    // code
}
```

This means the function is not returning any value.

It's also possible to return a value from a function. For this, we need to specify the `returnType` of the function during function declaration.

Then, the `return` statement can be used to return a value from a function.

For example,

```
int add (int a, int b) {
    return (a + b);
}
```



Here, we have the [data type](#) `int` instead of `void`. This means that the function returns an `int` value.

The code `return (a + b);` returns the sum of the two parameters as the function value.

The `return` statement denotes that the function has ended. Any code after `return` inside the function is not executed.

## Example 3: Add Two Numbers

```
// program to add two numbers using a function
```

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

// declaring a function
int add(int a, int b) {
    return (a + b);
}

int main() {

    int sum;

    // calling the function and storing
    // the returned value in sum
    sum = add(100, 78);

    cout << "100 + 78 = " << sum << endl;

    return 0;
}
```

[Run Code](#)

## Output

```
100 + 78 = 178
```

In the above program, the `add()` function is used to find the sum of two numbers.

We pass two `int` literals `100` and `78` while calling the function.

We store the returned value of the function in the variable `sum`, and then we print it.

```
#include<iostream>

int add(int a, int b) {
    return (a + b);
}

int main() {
    int sum;

    sum = add(100, 78);
    ... ..
}
```

The diagram illustrates the execution flow. A blue box labeled 'function call' has two arrows. One arrow points from the `add(100, 78);` line in `main()` to the `return (a + b);` line in the `add()` function. The other arrow points from the `return (a + b);` line back to the `sum =` part of the `sum = add(100, 78);` line in `main()`, indicating the return value is stored in `sum`.

Working of C++ Function with return

statement

Notice that `sum` is a variable of `int` type. This is because the return value of `add()` is of `int` type.

## Function Prototype

In C++, the code of function declaration should be before the function call. However, if we want to define a function after the function call, we need to use the function prototype. For example,

```
// function prototype
void add(int, int);

int main() {
    // calling the function before declaration.
    add(5, 3);
    return 0;
}

// function definition
void add(int a, int b) {
    cout << (a + b);
}
```

In the above code, the function prototype is:

```
void add(int, int);
```

This provides the compiler with information about the function name and its parameters. That's why we can use the code to call a function before the function has been defined.

The syntax of a function prototype is:

```
returnType functionName(dataType1, dataType2, ...);
```

## Example 4: C++ Function Prototype

```
// using function definition after main() function
// function prototype is declared before main()
```

```
#include <iostream>
```

```
using namespace std;
```

```
// function prototype
```

```
int add(int, int);
```

```
int main() {
```

```
    int sum;
```

```
    // calling the function and storing
```

```
    // the returned value in sum
```

```
    sum = add(100, 78);
```

```
    cout << "100 + 78 = " << sum << endl;
```

```
    return 0;
```

```
}
```

```
// function definition
```

```
int add(int a, int b) {
```

```
    return (a + b);
```

```
}
```

[Run Code](#)

## Output

```
100 + 78 = 178
```

The above program is nearly identical to **Example 3**. The only difference is that here, the function is defined **after** the function call.

That's why we have used a function prototype in this example.

## Benefits of Using User-Defined Functions

- Functions make the code reusable. We can declare them once and use them multiple times.
- Functions make the program easier as each small task is divided into a function.
- Functions increase readability.

## C++ Library Functions

Library functions are the built-in functions in C++ programming.

Programmers can use library functions by invoking the functions directly; they don't need to write the functions themselves.

Some common library functions in C++ are [sqrt\(\)](#), [abs\(\)](#), [isdigit\(\)](#), etc.

In order to use library functions, we usually need to include the header file in which these library functions are defined.

For instance, in order to use mathematical functions such as `sqrt()` and `abs()`, we need to include the header file [cmath](#).

## Example 5: C++ Program to Find the Square Root of a Number

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

int main() {
    double number, squareRoot;

    number = 25.0;

    // sqrt() is a library function to calculate the square root
    squareRoot = sqrt(number);

    cout << "Square root of " << number << " = " << squareRoot;

    return 0;
}
```

[Run Code](#)

## Output

Square root of 25 = 5

In this program, the `sqrt()` library function is used to calculate the square root of a number.

The function declaration of `sqrt()` is defined in the `cmath` header file. That's why we need to use the code `#include <cmath>` to use the `sqrt()` function.

### Also Read:

- [C++ Standard Library functions.](#)
- [C++ User-defined Function Types](#)

# C++ Inline Functions

In C++, we can declare a function as inline. This copies the function to the location of the function call in compile-time and may make the program execution faster.

Before following this tutorial,

## Inline Functions

To create an inline function, we use the `inline` keyword. For example,

```
inline returnType functionName(parameters) {  
    // code  
}
```

Notice the use of keyword `inline` before the function definition.

## C++ Inline Function

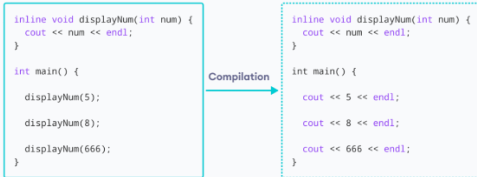
```
#include <iostream>  
using namespace std;  
  
inline void displayNum(int num) {  
    cout << num << endl;  
}  
  
int main() {  
    // first function call  
    displayNum(5);  
  
    // second function call  
    displayNum(8);  
  
    // third function call  
    displayNum(666);  
  
    return 0;  
}
```

[Run Code](#)

### Output

```
5  
8  
666
```

Here is how this program works:



Working of inline functions in C++

Here, we created an inline function named `displayNum()` that takes a single integer as a parameter.

We then called the function 3 times in the `main()` function with different arguments. Each time `displayNum()` is called, the compiler copies the code of the function to that call location.

## C++ Arrays

In C++, an array is a variable that can store multiple values of the same type. For example,

Suppose a class has **27** students, and we need to store all their grades. Instead of creating **27** separate variables, we can simply create an array:

```
double grade[27];
```

Here, `grade` is an array that can hold a maximum of **27** elements of `double` type.

In C++, the size and type of arrays cannot be changed after its declaration.

## C++ Array Declaration

```
dataType arrayName[arraySize];
```

For example,

```
int x[6];
```

Here,

- `int` - type of element to be stored
- `x` - name of the array
- `6` - size of the array

## Access Elements in C++ Array

In C++, each element in an array is associated with a number. The number is known as an **array index**. We can access elements of an array by using those indices.

```
// syntax to access array elements  
array[index];
```

Consider the array `x` we have seen above.

### Few Things to Remember:

- The array indices start with **0**. Meaning `x[0]` is the first element stored at index **0**.
- If the size of an array is `n`, the last element is stored at index `(n-1)`. In this example, `x[5]` is the last element.
- Elements of an array have consecutive addresses.

For example, suppose the starting address of `x[0]` is **2120**.

Then, the address of the next element `x[1]` will be **2124**, the address of `x[2]` will be **2128**, and so on.

Here, the size of each element is increased by **4**. This is because the size of `int` is **4** bytes.

## C++ Array Initialization



In C++, it's possible to initialize an array during declaration. For example,

```
// declare and initialize an array
int x[6] = {19, 10, 8, 17, 9, 15};
```

Another method to initialize array during declaration:

```
// declare and initialize an array
int x[] = {19, 10, 8, 17, 9, 15};
```

Here, we have not mentioned the size of the array. In such cases, the compiler automatically computes the size.

## C++ Array With Empty Members

In C++, if an array has a size `n`, we can store up to `n` number of elements in the array. However, what will happen if we store less than `n` number of elements.

For example,

```
// store only 3 elements in the array
int x[6] = {19, 10, 8};
```

Here, the array `x` has a size of `6`. However, we have initialized it with only `3` elements.

In such cases, the compiler assigns random values to the remaining places. Often, this random value is simply `0`.

## How to Insert and Print Array Elements?

```
int mark[5] = {19, 10, 8, 17, 9}
```

```
// change 4th element to 9
mark[3] = 9;
```

```
// take input from the user
// store the value at third position
cin >> mark[2];

// take input from the user
// insert at ith position
cin >> mark[i-1];

// print first element of the array
cout << mark[0];

// print ith element of the array
cout >> mark[i-1];
```

## Example 1: Displaying Array Elements

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

int main() {

    int numbers[5] = {7, 5, 6, 12, 35};

    cout << "The numbers are: ";

    // Printing array elements
    // using range based for loop
    for (int n : numbers) {
        cout << n << " ";
    }

    cout << "\nThe numbers are: ";

    // Printing array elements
    // using traditional for loop
    for (int i = 0; i < 5; ++i) {
        cout << numbers[i] << " ";
    }
}
```

```
    return 0;
}
```

[Run Code](#)

## Output

```
The numbers are: 7  5  6  12  35
The numbers are: 7  5  6  12  35
```

Here, we have used a [for loop](#) to iterate from `i = 0` to `i = 4`. In each iteration, we have printed `numbers[i]`.

We again used a range-based `for` loop to print out the elements of the array. To learn more about this loop, check [C++ Ranged for Loop](#).

## Example 2: Take Inputs from User and Store Them in an Array

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

int main() {

    int numbers[5];

    cout << "Enter 5 numbers: " << endl;

    // store input from user to array
    for (int i = 0; i < 5; ++i) {
        cin >> numbers[i];
    }

    cout << "The numbers are: ";

    // print array elements
    for (int n = 0; n < 5; ++n) {
        cout << numbers[n] << " ";
    }

    return 0;
}
```

[Run Code](#)

## Output

```
Enter 5 numbers:
11
12
13
14
15
The numbers are: 11 12 13 14 15
```

Once again, we have used a `for` loop to iterate from `i = 0` to `i = 4`. In each iteration, we took input from the user and stored it in `numbers[i]`.

Then, we used another `for` loop to print all the array elements.

## C++ Array Out of Bounds

If we declare an array of size **10**, then the array will contain elements from index **0** to **9**.

However, if we try to access the element at index **10** or more than **10**, it will result in an undefined behavior.

This type of error is likely when we use a normal for loop and we access the array element using the `[]` operator.

**Note:** To overcome this type of error we should prefer [range based for loop](#) where no such operators are needed to access the array elements.

## Example 3: Display Sum and Average of Array Elements Using for Loop

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

int main() {
```

```

// initialize an array without specifying size
double numbers[] = {7, 5, 6, 12, 35, 27};

double sum = 0;
double count = 0;
double average;

cout << "The numbers are: ";

// print array elements
// use of range-based for loop
for (const double &n : numbers) {
    cout << n << " ";

    // calculate the sum
    sum += n;

    // count the no. of array elements
    ++count;
}

// print the sum
cout << "\nTheir Sum = " << sum << endl;

// find the average
average = sum / count;
cout << "Their Average = " << average << endl;

return 0;
}

```

[Run Code](#)

## Output

```

The numbers are: 7  5  6  12  35  27
Their Sum = 92
Their Average = 15.3333

```

In this program:

1. We have initialized a `double` array named `numbers` but without specifying its size. We also declared three `double` variables: `sum`, `count`, and `average`. Here, `sum = 0` and `count = 0`.
2. Then we used a range-based `for` loop to print the array elements. In each iteration of the loop, we add the current array element to `sum`.
3. We also increase the value of `count` by `1` in each iteration, so that we can get the size of the array by the end of the for loop.
4. After printing all the elements, we print the sum and the average of all the numbers. The average of the numbers is given by `average = sum / count;`

# Passing Array to a Function in C++ Programming

In C++, we can pass arrays as an argument to a function. And, also we can return arrays from a function.

Before you learn about passing arrays as a function argument, make sure you know about [C++ Arrays](#) and [C++ Functions](#).

## Syntax

The syntax for passing an array to a function is:

```
returnType functionName(dataType arrayName[]) {  
    // code  
}
```

## Example 1: Passing One-dimensional Array to a Function

```
// C++ Program to display marks of 5 students

#include <iostream>
using namespace std;

const int ARRAY_SIZE = 5;

// declare function to display marks
// take a 1d array as parameter
void display(int m[]) {
    cout << "Displaying marks: " << endl;

    // display array elements
    for (int i = 0; i < ARRAY_SIZE; ++i) {
        cout << "Student " << i + 1 << ": " << m[i] << endl;
    }
}

int main() {

    // declare and initialize an array
    int marks[ARRAY_SIZE] = {88, 76, 90, 61, 69};

    // call display function
    // pass array as argument
    display(marks);

    return 0;
}
Run Code
```

## Output

```
Displaying marks:
Student 1: 88
Student 2: 76
Student 3: 90
Student 4: 61
Student 5: 69
```

Here,

1. When we call a function by passing an array as the argument, only the name of the array is used.

```
display(marks);
```

Here, the argument `marks` represent the memory address of the first element of array `marks[5]`.

2. However, notice the parameter of the `display()` function.

```
void display(int m[])
```

Here, we are just expecting an array of integers. C++ handles passing an array to a function in this way to save memory and time.

## Passing Multidimensional Array to a Function

We can also pass [Multidimensional arrays](#) as an argument to the function. For example,

### Example 2: Passing Multidimensional Array to a Function

```
// C++ Program to display the elements of two
// dimensional array by passing it to a function

#include <iostream>
using namespace std;

// define a function
// pass a 2d array as a parameter
void display(int n[][2]) {
    cout << "Displaying Values: " << endl;
    for (int i = 0; i < 3; ++i) {
        for (int j = 0; j < 2; ++j) {
            cout << "num[" << i << "][" << j << "]: " << n[i][j] << endl;
        }
    }
}
```



```
    }  
}  
  
int main() {  
  
    // initialize 2d array  
    int num[3][2] = {  
        {3, 4},  
        {9, 5},  
        {7, 1}  
    };  
  
    // call the function  
    // pass a 2d array as an argument  
    display(num);  
  
    return 0;  
}  
Run Code
```

## Output

```
Displaying Values:  
num[0][0]: 3  
num[0][1]: 4  
num[1][0]: 9  
num[1][1]: 5  
num[2][0]: 7  
num[2][1]: 1
```

In the above program, we have defined a function named `display()`. The function takes a two dimensional array, `int n[][2]` as its argument and prints the elements of the array.

While calling the function, we only pass the name of the two dimensional array as the function argument `display(num)`.

**Note:** It is not mandatory to specify the number of rows in the array. However, the number of columns should always be specified. This is why we have used `int n[][2]`.

We can also pass arrays with more than 2 dimensions as a function argument.

## C++ Multidimensional Arrays

In C++, we can create an [array](#) of an array, known as a multidimensional array. For example:

```
int x[3][4];
```

Here, `x` is a two-dimensional array. It can hold a maximum of 12 elements. We can think of this array as a table with 3 rows and each row has 4 columns as shown below.

	Col 1	Col 2	Col 3	Col 4
Row 1	<code>x[0][0]</code>	<code>x[0][1]</code>	<code>x[0][2]</code>	<code>x[0][3]</code>
Row 2	<code>x[1][0]</code>	<code>x[1][1]</code>	<code>x[1][2]</code>	<code>x[1][3]</code>
Row 3	<code>x[2][0]</code>	<code>x[2][1]</code>	<code>x[2][2]</code>	<code>x[2][3]</code>

Elements in two-dimensional array in C++ Programming

Three-dimensional arrays also work in a similar way. For example:

```
float x[2][4][3];
```

This array `x` can hold a maximum of 24 elements.

We can find out the total number of elements in the array simply by multiplying its dimensions:

$$2 \times 4 \times 3 = 24$$

## Multidimensional Array Initialization

Like a normal array, we can initialize a multidimensional array in more than one way.

### 1. Initialization of two-dimensional array

```
int test[2][3] = {2, 4, 5, 9, 0, 19};
```

The above method is not preferred. A better way to initialize this array with the same array elements is given below:

```
int test[2][3] = { {2, 4, 5}, {9, 0, 19}};
```

This array has 2 rows and 3 columns, which is why we have two rows of elements with 3 elements each.

	Col 1	Col 2	Col 3
Row 1	2	4	5
Row 2	9	0	19

Initializing a two-dimensional array in C++

### 2. Initialization of three-dimensional array

```
int test[2][3][4] = {3, 4, 2, 3, 0, -3, 9, 11, 23, 12, 23,  
2, 13, 4, 56, 3, 5, 9, 3, 5, 5, 1, 4, 9};
```

This is not a good way of initializing a three-dimensional array. A better way to initialize this array is:

```
int test[2][3][4] = {  
    { {3, 4, 2, 3}, {0, -3, 9, 11}, {23, 12, 23, 2} },  
    { {13, 4, 56, 3}, {5, 9, 3, 5}, {5, 1, 4, 9} }  
};
```

Notice the dimensions of this three-dimensional array.

The first dimension has the value `2`. So, the two elements comprising the first dimension are:

```
Element 1 = { {3, 4, 2, 3}, {0, -3, 9, 11}, {23, 12, 23, 2} }  
Element 2 = { {13, 4, 56, 3}, {5, 9, 3, 5}, {5, 1, 4, 9} }
```

The second dimension has the value `3`. Notice that each of the elements of the first dimension has three elements each:

```
{3, 4, 2, 3}, {0, -3, 9, 11} and {23, 12, 23, 2} for Element 1.  
{13, 4, 56, 3}, {5, 9, 3, 5} and {5, 1, 4, 9} for Element 2.
```

Finally, there are four `int` numbers inside each of the elements of the second dimension:

```
{3, 4, 2, 3}  
{0, -3, 9, 11}  
... ..  
... ..
```

## Example 1: Two Dimensional Array

```
// C++ Program to display all elements  
// of an initialised two dimensional array  
  
#include <iostream>  
using namespace std;
```

```

int main() {
    int test[3][2] = {{2, -5},
                      {4, 0},
                      {9, 1}};

    // use of nested for loop
    // access rows of the array
    for (int i = 0; i < 3; ++i) {

        // access columns of the array
        for (int j = 0; j < 2; ++j) {
            cout << "test[" << i << "][" << j << "] = " << test[i][j] << endl;
        }
    }

    return 0;
}

```

[Run Code](#)

## Output

```

test[0][0] = 2
test[0][1] = -5
test[1][0] = 4
test[1][1] = 0
test[2][0] = 9
test[2][1] = 1

```

In the above example, we have initialized a two-dimensional `int` array named `test` that has 3 "rows" and 2 "columns".

Here, we have used the nested `for` loop to display the array elements.

- the outer loop from `i == 0` to `i == 2` access the rows of the array
- the inner loop from `j == 0` to `j == 1` access the columns of the array

Finally, we print the array elements in each iteration.

## Example 2: Taking Input for Two Dimensional Array

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

int main() {
    int numbers[2][3];

    cout << "Enter 6 numbers: " << endl;

    // Storing user input in the array
    for (int i = 0; i < 2; ++i) {
        for (int j = 0; j < 3; ++j) {
            cin >> numbers[i][j];
        }
    }

    cout << "The numbers are: " << endl;

    // Printing array elements
    for (int i = 0; i < 2; ++i) {
        for (int j = 0; j < 3; ++j) {
            cout << "numbers[" << i << "][" << j << "]: " << numbers[i][j] <<
endl;
        }
    }

    return 0;
}
```

[Run Code](#)

## Output

```
Enter 6 numbers:
1
2
3
4
5
6
The numbers are:
numbers[0][0]: 1
numbers[0][1]: 2
numbers[0][2]: 3
numbers[1][0]: 4
```

```
numbers[1][1]: 5
numbers[1][2]: 6
```

Here, we have used a [nested for loop](#) to take the input of the 2d array. Once all the input has been taken, we have used another nested `for` loop to print the array members.

## Example 3: Three Dimensional Array

```
// C++ Program to Store value entered by user in
// three dimensional array and display it.

#include <iostream>
using namespace std;

int main() {
    // This array can store upto 12 elements (2x3x2)
    int test[2][3][2] = {
        {
            {1, 2},
            {3, 4},
            {5, 6}
        },
        {
            {7, 8},
            {9, 10},
            {11, 12}
        }
    };

    // Displaying the values with proper index.
    for (int i = 0; i < 2; ++i) {
        for (int j = 0; j < 3; ++j) {
            for (int k = 0; k < 2; ++k) {
                cout << "test[" << i << "][" << j << "][" << k << "] = " <<
test[i][j][k] << endl;
            }
        }
    }

    return 0;
}
```

```
}  
Run Code
```

## Output

```
test[0][0][0] = 1  
test[0][0][1] = 2  
test[0][1][0] = 3  
test[0][1][1] = 4  
test[0][2][0] = 5  
test[0][2][1] = 6  
test[1][0][0] = 7  
test[1][0][1] = 8  
test[1][1][0] = 9  
test[1][1][1] = 10  
test[1][2][0] = 11  
test[1][2][1] = 12
```

The basic concept of printing elements of a 3d array is similar to that of a 2d array.

However, since we are manipulating 3 dimensions, we use a nested for loop with 3 total loops instead of just 2:

- the outer loop from `i == 0` to `i == 1` accesses the first dimension of the array
- the middle loop from `j == 0` to `j == 2` accesses the second dimension of the array
- the innermost loop from `k == 0` to `k == 1` accesses the third dimension of the array

As we can see, the complexity of the array increases exponentially with the increase in dimensions.

## C++ OOPs Concepts

The major purpose of C++ programming is to introduce the concept of object orientation to the C programming language.



Object Oriented Programming is a paradigm that provides many concepts such as **inheritance, data binding, polymorphism etc.**

The programming paradigm where everything is represented as an object is known as truly object-oriented programming language. **Smalltalk** is considered as the first truly object-oriented programming language.

## OOPs (Object Oriented Programming System)

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

- Object
- Class
- Inheritance
- Polymorphism
- Abstraction
- Encapsulation

### Object

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

### Class

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

A Class in C++ is the foundational element that leads to Object-Oriented programming. A class instance must be created in order to access and use the user-defined data type's data members and member functions. An object's class acts as its blueprint. Take the class of cars as an example. Even if different names and brands may be used for different cars, all of them will have some characteristics in common, such as four wheels, a speed limit, a range of miles, etc. In this case, the class of car is represented by the wheels, the speed limitations, and the mileage.

### Inheritance

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

1. Sub class - Subclass or Derived Class refers to a class that receives properties from another class.
2. Super class - The term "Base Class" or "Super Class" refers to the class from which a subclass inherits its properties.
3. Reusability - As a result, when we wish to create a new class, but an existing class already contains some of the code we need, we can generate our new class from the old class thanks to inheritance. This allows us to utilize the fields and methods of the pre-existing class.

## Polymorphism

When **one task is performed by different ways** i.e. known as polymorphism. For example: to convince the customer differently, to draw something e.g. shape or rectangle etc.

Different situations may cause an operation to behave differently. The type of data utilized in the operation determines the behavior.

## Abstraction

**Hiding internal details and showing functionality** is known as abstraction. Data abstraction is the process of exposing to the outside world only the information that is absolutely necessary while concealing implementation or background information. For example: phone call, we don't know the internal processing.

In C++, we use abstract class and interface to achieve abstraction.

## Encapsulation

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

Encapsulation is typically understood as the grouping of related pieces of information and data into a single entity. Encapsulation is the process of tying together data and the functions that work with it in object-oriented programming.

**Dynamic Binding** - In dynamic binding, a decision is made at runtime regarding the code that will be run in response to a function call. For this, C++ supports virtual functions.

## Advantage of OOPs over Procedure-oriented programming language

1. OOPs makes development and maintenance easier where as in Procedure-oriented programming language it is not easy to manage if code grows as project size grows.
2. OOPs provide data hiding whereas in Procedure-oriented programming language a global data can be accessed from anywhere.
3. OOPs provide ability to simulate real-world event much more effectively. We can provide the solution of real word problem if we are using the Object-Oriented Programming language.

## Why do we need oops in C++?

There were various drawbacks to the early methods of programming, as well as poor performance. The approach couldn't effectively address real-world issues since, similar to procedural-oriented programming, you couldn't reuse the code within the program again, there was a difficulty with global data access, and so on.

With the use of classes and objects, object-oriented programming makes code maintenance simple. Because inheritance allows for code reuse, the program is simpler because you don't have to write the same code repeatedly. Data hiding is also provided by ideas like encapsulation and abstraction.

## Why is C++ a partial oop?

The object-oriented features of the C language were the primary motivation behind the construction of the C++ language.

The C++ programming language is categorized as a partial object-oriented programming language despite the fact that it supports OOP concepts, including classes, objects, inheritance, encapsulation, abstraction, and polymorphism.

1) The main function must always be outside the class in C++ and is required. This means that we may do without classes and objects and have a single main function in the application

It is expressed as an object in this case, which is the first time Pure OOP has been violated.

2) Global variables are a feature of the C++ programming language that can be accessed by any other object within the program and are defined outside of it. Encapsulation is broken here. Even though C++ encourages encapsulation for classes and objects, it ignores it for global variables.

## Overloading

Polymorphism also has a subset known as overloading. An existing operator or function is said to be overloaded when it is forced to operate on a new data type.

## Conclusion

You will have gained an understanding of the need for object-oriented programming, what C++ OOPs are, and the fundamentals of OOPs, such as polymorphism, inheritance, encapsulation, etc., after reading this course on OOPS Concepts in C++. Along with instances of polymorphism and inheritance, you also learned about the benefits of C++ OOPs.

## C++ Object and Class

Since C++ is an object-oriented language, program is designed using objects and classes in C++.

### C++ Object

In C++, Object is a real world entity, for example, chair, car, pen, mobile, laptop etc.

In other words, object is an entity that has state and behavior. Here, state means data and behavior means functionality.

Object is a runtime entity, it is created at runtime.

Object is an instance of a class. All the members of the class can be accessed through object.

Let's see an example to create object of student class using s1 as the reference variable.

1. Student s1; *//creating an object of Student*

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

---

## C++ Class

In C++, class is a group of similar objects. It is a template from which objects are created. It can have fields, methods, constructors etc.

Let's see an example of C++ class that has three fields only.

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

## C++ Object and Class Example

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

1. **#include** <iostream>
2. **using namespace** std;
3. **class** Student {
4.     **public:**
5.     **int** id;*//data member (also instance variable)*
6.     string name;*//data member(also instance variable)*

```

7. };
8. int main() {
9.     Student s1; //creating an object of Student
10.    s1.id = 201;
11.    s1.name = "Sonoo Jaiswal";
12.    cout<<s1.id<<endl;
13.    cout<<s1.name<<endl;
14.    return 0;
15.}

```

Output:

```

201
Sonoo Jaiswal

```

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

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

```

1. #include <iostream>
2. using namespace std;
3. class Student {
4.     public:
5.         int id;//data member (also instance variable)
6.         string name;//data member(also instance variable)
7.         void insert(int i, string n)
8.         {
9.             id = i;
10.            name = n;
11.        }
12.        void display()
13.        {
14.            cout<<id<<" "<<name<<endl;

```

```

15.     }
16. };
17. int main(void) {
18.     Student s1; //creating an object of Student
19.     Student s2; //creating an object of Student
20.     s1.insert(201, "Sonoo");
21.     s2.insert(202, "Nakul");
22.     s1.display();
23.     s2.display();
24.     return 0;
25. }

```

Output:

```

201  Sonoo
202  Nakul

```

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

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

```

1. #include <iostream>
2. using namespace std;
3. class Employee {
4.     public:
5.         int id;//data member (also instance variable)
6.         string name;//data member(also instance variable)
7.         float salary;
8.         void insert(int i, string n, float s)
9.         {
10.             id = i;
11.             name = n;
12.             salary = s;

```

```

13.     }
14.     void display()
15.     {
16.         cout<<id<<" "<<name<<" "<<salary<<endl;
17.     }
18. };
19. int main(void) {
20.     Employee e1; //creating an object of Employee
21.     Employee e2; //creating an object of Employee
22.     e1.insert(201, "Sonoo", 990000);
23.     e2.insert(202, "Nakul", 29000);
24.     e1.display();
25.     e2.display();
26.     return 0;
27. }

```

Output:

```

201  Sonoo  990000
202  Nakul  29000

```

## C++ Constructor

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

In brief, A particular procedure called a constructor is called automatically when an object is created in C++. In general, it is employed to create the data members of new things. In C++, the class or structure name also serves as the constructor name. When an object is completed, the constructor is called. Because it creates the values or gives data for the thing, it is known as a constructor.

The Constructors prototype looks like this:

1. <class-name> (list-of-parameters);

The following syntax is used to define the class's constructor:



1. `<class-name> (list-of-parameters) { // constructor definition }`

The following syntax is used to define a constructor outside of a class:

1. `<class-name>::<class-name> (list-of-parameters){ // constructor definition}`

Constructors lack a return type since they don't have a return value.

There can be two types of constructors in C++.

- Default constructor
- Parameterized constructor

## C++ Default Constructor

A constructor which has no argument is known as default constructor. It is invoked at the time of creating object.

Let's see the simple example of C++ default Constructor.

```
1. #include <iostream>
2. using namespace std;
3. class Employee
4. {
5.     public:
6.         Employee()
7.         {
8.             cout<<"Default Constructor Invoked"<<endl;
9.         }
10. };
11. int main(void)
12. {
13.     Employee e1; //creating an object of Employee
14.     Employee e2;
15.     return 0;
16. }
```

## Output:

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

## C++ Parameterized Constructor

A constructor which has parameters is called parameterized constructor. It is used to provide different values to distinct objects.

Let's see the simple example of C++ Parameterized Constructor.

```
1. #include <iostream>
2. using namespace std;
3. class Employee {
4.     public:
5.         int id;//data member (also instance variable)
6.         string name;//data member(also instance variable)
7.         float salary;
8.         Employee(int i, string n, float s)
9.         {
10.            id = i;
11.            name = n;
12.            salary = s;
13.        }
14.        void display()
15.        {
16.            cout<<id<<" "<<name<<" "<<salary<<endl;
17.        }
18.};
19. int main(void) {
20.    Employee e1 =Employee(101, "Sonoo", 890000); //creating an object of Employee
21.    Employee e2=Employee(102, "Nakul", 59000);
22.    e1.display();
23.    e2.display();
24.    return 0;
25.}
```

## Output:

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

## What distinguishes constructors from a typical member function?

1. Constructor's name is the same as the class's
2. Default There isn't an input argument for constructors. However, input arguments are available for copy and parameterized constructors.
3. There is no return type for constructors.
4. An object's constructor is invoked automatically upon creation.
5. It must be shown in the classroom's open area.
6. The C++ compiler creates a default constructor for the object if a constructor is not specified (expects any parameters and has an empty body).

By using a practical example, let's learn about the various constructor types in C++.

## What are the characteristics of a constructor?

1. The constructor has the same name as the class it belongs to.
2. Although it is possible, constructors are typically declared in the class's public section. However, this is not a must.
3. Because constructors don't return values, they lack a return type.
4. When we create a class object, the constructor is immediately invoked.
5. Overloaded constructors are possible.
6. Declaring a constructor virtual is not permitted.
7. One cannot inherit a constructor.
8. Constructor addresses cannot be referenced to.
9. When allocating memory, the constructor makes implicit calls to the new and delete operators.

## What is a copy constructor?

A member function known as a copy constructor initializes an item using another object from the same class-an in-depth discussion on Copy Constructors.

Every time we specify one or more non-default constructors (with parameters) for a class, we also need to include a default constructor (without parameters), as the compiler won't supply one in this circumstance. The best practice is to always declare a default constructor, even though it is not required.

A reference to an object belonging to the same class is required by the copy constructor.

1. Sample(Sample &t)
2. {
3. id=t.id;
4. }

## What is a destructor in C++?

An equivalent special member function to a constructor is a destructor. The constructor

1. <class-name>()
2. {
3. }

The language used to define the class's destructor outside of it

1. <class-name>::~ ~ <class-name>(){}

## C++ Destructor

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

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

**Note: C++ destructor cannot have parameters. Moreover, modifiers can't be applied on destructors.**

## C++ Constructor and Destructor Example

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

```
1. #include <iostream>
2. using namespace std;
3. class Employee
4. {
5.     public:
6.         Employee()
7.         {
8.             cout<<"Constructor Invoked"<<endl;
9.         }
10.        ~Employee()
11.        {
12.            cout<<"Destructor Invoked"<<endl;
13.        }
14. };
15. int main(void)
16. {
17.     Employee e1; //creating an object of Employee
18.     Employee e2; //creating an object of Employee
19.     return 0;
20. }
```

Output:

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

## C++ Inheritance

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

In C++, the class which inherits the members of another class is called derived class and the class whose members are inherited is called base class. The derived class is the specialized class for the base class.

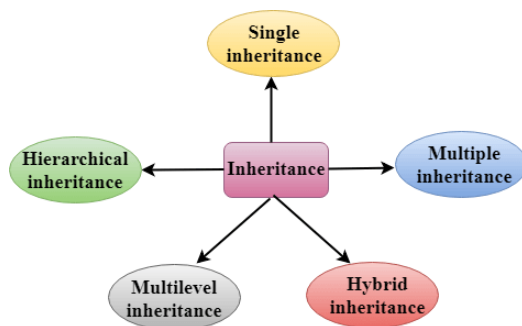
## Advantage of C++ Inheritance

**Code reusability:** Now you can reuse the members of your parent class. So, there is no need to define the member again. So less code is required in the class.

## Types Of Inheritance

**C++ supports five types of inheritance:**

- Single inheritance
- Multiple inheritance
- Hierarchical inheritance
- Multilevel inheritance
- Hybrid inheritance



## Derived Classes

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

The Syntax of Derived class:

1. **class** derived\_class\_name :: visibility-mode base\_class\_name

```
2. {  
3.     // body of the derived class.  
4. }
```

**Where,**

**derived\_class\_name:** It is the name of the derived class.

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

**base\_class\_name:** It is the name of the base class.

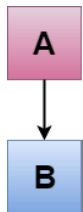
- When the base class is privately inherited by the derived class, public members of the base class becomes the private members of the derived class. Therefore, the public members of the base class are not accessible by the objects of the derived class only by the member functions of the derived class.

**Note:**

- In C++, the default mode of visibility is private.
- The private members of the base class are never inherited.

## C++ Single Inheritance

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



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

## C++ Single Level Inheritance Example: Inheriting Fields

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

```
1. #include <iostream>
2. using namespace std;
3. class Account {
4.     public:
5.     float salary = 60000;
6. };
7. class Programmer: public Account {
8.     public:
9.     float bonus = 5000;
10. };
11. int main(void) {
12.     Programmer p1;
13.     cout<<"Salary: "<<p1.salary<<endl;
14.     cout<<"Bonus: "<<p1.bonus<<endl;
15.     return 0;
16. }
```

Output:

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

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

## 1. Single Inheritance

In single inheritance, a derived class inherits from a single base class.

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

class Base {
public:
    void show() {
        cout << "Base class show function." << endl;
    }
};
```



```

    }
};
class Derived : public Base {
public:
    void display() {
        cout << "Derived class display function." << endl;
    }
};

int main() {
    Derived obj;
    obj.show();
    obj.display();
    return 0;
}

```

### Output:

```

Base class show function.
Derived class display function.

```

## 2. Multiple Inheritance

In multiple inheritance, a derived class inherits from more than one base class.

```

• #include <iostream>
using namespace std;

// Base class 1
class Base1 {
public:
    void method1() {
        cout << "Method of Base1 class" << endl;
    }
};

// Base class 2
class Base2 {
public:
    void method2() {
        cout << "Method of Base2 class" << endl;
    }
}

```

```

};

// Derived class inheriting from Base1 and Base2

class Derived : public Base1, public Base2 {
public:
    void derivedMethod() {

        cout << "Method of Derived class" << endl;
    }
};

int main() {
    Derived obj;
    obj.method1();    // Method from Base1 class
    obj.method2();    // Method from Base2 class
    obj.derivedMethod(); // Method from Derived class
    return 0;
}

```

### Output:

```

Method of Base1 class
Method of Base2 class
Method of Derived class

```

**Explanation:** The `Derived` class inherits methods from both `Base1` and `Base2`, allowing it to use methods from both base classes.

## • Multilevel Inheritance

In multilevel inheritance, a class is derived from another derived class.

### Code Example:

```

#include <iostream>
using namespace std;

// Base class
class Base {
public:
    void baseMethod() {
        cout << "Method of Base class" << endl;
    }
};

// Derived class inheriting from Base

class Intermediate : public Base {

```

```

public:

    void intermediateMethod() {

        cout << "Method of Intermediate class" << endl;
    }
};

// Further derived class inheriting from Intermediate
class Derived : public Intermediate {

public:

    void derivedMethod() {
        cout << "Method of Derived class" << endl;
    }
};

int main() {
    Derived obj;
    obj.baseMethod();           // Method from Base class
    obj.intermediateMethod();    // Method from Intermediate class
    obj.derivedMethod();         // Method from Derived class
    return 0;
}

```

### Output:

```

Method of Base class
Method of Intermediate class
Method of Derived class

```

## 3. Multilevel Inheritance

In multilevel inheritance, a class is derived from another derived class.

```

#include <iostream>
using namespace std;

class Base {
public:
    void show() {
        cout << "Base class show function." << endl;
    }
};

```

```

class Intermediate : public Base {
public:
    void display() {
        cout << "Intermediate class display function." << endl;
    }
};

class Derived : public Intermediate {
public:
    void showDerived() {

        cout << "Derived class showDerived function." << endl;
    }
};

int main() {
    Derived obj;
    obj.show();
    obj.display();
    obj.showDerived();
    return 0;
}

```

### Output:

```

Base class show function.
Intermediate class display function.
Derived class showDerived function.

```

## 4. Hierarchical Inheritance

In hierarchical inheritance, multiple derived classes inherit from a single base class.

```

#include <iostream>
using namespace std;

class Base {
public:
    void show() {

```

```

        cout << "Base class show function." << endl;
    }
};

class Derived1 : public Base {
public:
    void display1() {
        cout << "Derived1 class display1 function." << endl;
    }
};

class Derived2 : public Base {
public:
    void display2() {
        cout << "Derived2 class display2 function." << endl;
    }
};

int main() {
    Derived1 obj1;
    Derived2 obj2;

    obj1.show();
    obj1.display1();

    obj2.show();
    obj2.display2();

    return 0;
}

```

### Output:

```

Base class show function.
Derived1 class display1 function.
Base class show function.
Derived2 class display2 function.

```

### Hybrid Inheritance

Hybrid inheritance is a combination of two or more types of inheritance. It can be complex and is often avoided to prevent ambiguity.

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

class Base {
public:

    void show() {

        cout << "Base class show function." << endl;
    }
};

class Intermediate1 : public Base {

public:
    void display1() {
        cout << "Intermediate1 class display1 function." << endl;
    }
};

class Intermediate2 : public Base {
public:
    void display2() {

        cout << "Intermediate2 class display2 function." << endl;
    }
};

class Derived : public Intermediate1, public Intermediate2 {
public:
    void showDerived() {
        cout << "Derived class showDerived function." << endl;
    }
};

int main() {
    Derived obj;
    obj.Intermediate1::show(); // Ambiguity resolved by specifying the base
    class
```

```

        obj.display1();
        obj.Intermediate2::show(); // Ambiguity resolved by specifying the base
class
        obj.display2();
        obj.showDerived();

    return 0;
}

```

### Output:

```

Base class show function.
Intermediate1 class display1 function.
Base class show function.
Intermediate2 class display2 function.
Derived class showDerived function.

```

In this example of hybrid inheritance, ambiguity arises when calling the `show` function because Base class methods are inherited through multiple paths. The ambiguity is resolved by explicitly specifying which path to use (`Intermediate1::show` and `Intermediate2::show`).

## Encapsulation

Encapsulation is the practice of bundling data and methods that operate on the data within a single class, and restricting direct access to some of the class's components. This is achieved using private and public access specifiers.

```

#include <iostream>
using namespace std;

class Account {
private:
    double balance; // Private member variable

public:
    // Constructor to initialize balance
    Account(double initialBalance) : balance(initialBalance) {}

    // Public method to deposit money
    void deposit(double amount) {

```

```

        if (amount > 0) {
            balance += amount;
        }
    }

    // Public method to withdraw money
    void withdraw(double amount) {

        if (amount > 0 && amount <= balance) {
            balance -= amount;
        }
    }

    // Public method to get the current balance
    double getBalance() const {
        return balance;
    }
};

int main() {
    Account myAccount(1000.0);
    myAccount.deposit(500.0);
    myAccount.withdraw(200.0);
    cout << "Current Balance: $" << myAccount.getBalance() << endl;

    return 0;
}

```

### Output:

Current Balance: \$1300

### Notes:

- **Encapsulation** is achieved here by keeping the `balance` variable private and providing public methods (`deposit`, `withdraw`, `getBalance`) to access and modify it. This ensures that the balance can only be modified in controlled ways.



## Polymorphism

Polymorphism allows methods to perform different tasks based on the object that is calling them. In C++, this can be achieved using function overloading and virtual functions.

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

class Shape {
public:

    // Virtual function for polymorphism
    virtual void draw() const {

        cout << "Drawing a shape." << endl;
    }
};

class Circle : public Shape {

public:
    void draw() const override {
        cout << "Drawing a circle." << endl;

    }
};

class Square : public Shape {

public:
    void draw() const override {
        cout << "Drawing a square." << endl;
    }
};

int main() {
    Shape* shape1 = new Circle();
    Shape* shape2 = new Square();
```

```
    shape1->draw(); // Calls Circle's draw
    shape2->draw(); // Calls Square's draw
    delete shape1;
    delete shape2;
    return 0;
}
```

### Output:

```
Drawing a circle.
Drawing a square.
```

### Notes:

- **Polymorphism** is demonstrated by the `draw` function, which is declared as `virtual` in the base class `Shape`. This allows the `draw` function to be overridden in derived classes (`Circle` and `Square`).  
When `shape1->draw()` and `shape2->draw()` are called, the appropriate `draw` method is invoked based on the actual object type.

## Abstraction

Abstraction in C++ is a key concept in object-oriented programming that allows you to handle complexity by hiding unnecessary details from the user and showing only the essential features of an object. It can be implemented using abstract classes and interfaces

### Key Concepts

1. **Abstract Class:** A class that cannot be instantiated and usually contains at least one pure virtual function.

#### Pure Virtual Function:

A function declared in a base class that must be overridden in derived classes. It is declared by assigning 0 to the function in the base class.

In C++, a virtual function is a member function in a base class that you expect to override in derived classes. It enables dynamic (or runtime) polymorphism, allowing you to call derived class methods through base class pointers or references. This feature is crucial for achieving runtime flexibility in object-oriented programming.

## Key Concepts

1. **Dynamic Polymorphism:** Virtual functions enable dynamic polymorphism by allowing the appropriate method to be called based on the actual object type pointed to or referenced, not the type of the pointer or reference.
2. **Virtual Table (vtable):** Each class with virtual functions has a virtual table (vtable), which is an internal mechanism used by C++ to resolve method calls at runtime. The vtable contains pointers to the virtual functions of the class.

## Example: Abstract Class and Pure Virtual Function

```
#include <iostream>

// Abstract class
class Shape {
public:
    // Pure virtual function
    virtual void draw() = 0;

    // Non-pure virtual function with a definition
    virtual void display() {
        std::cout << "Displaying a shape" << std::endl;
    }
};

// Derived class
class Circle : public Shape {
public:
    void draw() override {
        std::cout << "Drawing a circle" << std::endl;
    }

    void display() override {
        std::cout << "Displaying a circle" << std::endl;
    }
};
```

```

    }
};

// Another derived class
class Rectangle : public Shape {

public:
    void draw() override {
        std::cout << "Drawing a rectangle" << std::endl;
    }

    void display() override {
        std::cout << "Displaying a rectangle" << std::endl;
    }
};

int main() {
    // Shape shape;

    // This line would cause a compilation error because Shape is
    abstract

    // Pointers to the base class can point to derived class objects
    Shape* shape1 = new Circle();

    Shape* shape2 = new Rectangle();

    // Calling the overridden methods

    shape1->draw();    // Output: Drawing a circle
    shape1->display(); // Output: Displaying a circle

    shape2->draw();    // Output: Drawing a rectangle
    shape2->display(); // Output: Displaying a rectangle

    // Cleaning up

    delete shape1;
    delete shape2;

    return 0;
}

```

### *Output*

```

Drawing a circle
Displaying a circle
Drawing a rectangle
Displaying a rectangle

```

## Summary

- **Encapsulation:** Hides the internal state and requires all interactions to be performed through methods.
- **Polymorphism:** Allows methods to be implemented in multiple ways depending on the object type, achieved through virtual functions.
- **Abstraction:** Hides complex implementation details and exposes only the essential features, achieved through abstract classes and pure virtual functions

The four pillars of C++ (or object-oriented programming in general) are:

1. **Encapsulation:** The concept of bundling data (attributes) and methods (functions) that operate on the data into a single unit, known as a class. It also involves controlling access to the data via access specifiers (public, private, protected) to protect the integrity of the data.
2. **Inheritance:** The mechanism by which a new class (derived class) acquires the properties and behaviors (methods) of an existing class (base class). It promotes code reusability and establishes a hierarchical relationship between classes.
3. **Polymorphism:** The ability of different classes to be treated through a common interface, with the ability to use a single function or operator in different ways. This can be achieved through function overloading, operator overloading, and virtual functions.
4. **Abstraction:** The concept of hiding the complex implementation details and showing only the essential features of the object. It is achieved through abstract classes and interfaces, allowing a focus on what an object does rather than how it does it.

Pure Virtual Function: A pure virtual function is a virtual function that has no implementation in the base class and is intended to be overridden in derived classes. It makes the base class abstract, meaning you cannot instantiate it directly.

## C++ Single Level Inheritance Example: Inheriting Methods

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

```
1. #include <iostream>
2. using namespace std;
3. class Animal {
4.     public:
5.     void eat() {
6.         cout<<"Eating..."<<endl;
7.     }
8. };
9. class Dog: public Animal
10. {
11.     public:
12.     void bark(){
13.         cout<<"Barking...";
14.     }
15. };
16. int main(void) {
17.     Dog d1;
18.     d1.eat();
19.     d1.bark();
20.     return 0;
21. }
```

Output:

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

Let's see a simple example.

```
1. #include <iostream>
2. using namespace std;
3. class A
4. {
5.     int a = 4;
6.     int b = 5;
```

```

7.  public:
8.  int mul()
9.  {
10.     int c = a*b;
11.     return c;
12. }
13.};
14.
15. class B : private A
16. {
17.  public:
18.  void display()
19.  {
20.     int result = mul();
21.     std::cout << "Multiplication of a and b is : " << result << std::endl;
22. }
23.};
24. int main()
25. {
26.  B b;
27.  b.display();
28.
29.  return 0;
30. }

```

Output:

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

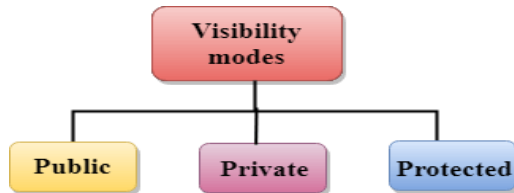
In the above example, class A is privately inherited. Therefore, the mul() function of class 'A' cannot be accessed by the object of class B. It can only be accessed by the member function of class B.

## How to make a Private Member Inheritable

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

C++ introduces a third visibility modifier, i.e., **protected**. The member which is declared as protected will be accessible to all the member functions within the class as well as the class immediately derived from it.

**Visibility modes can be classified into three categories:**



- **Public:** When the member is declared as public, it is accessible to all the functions of the program.
- **Private:** When the member is declared as private, it is accessible within the class only.
- **Protected:** When the member is declared as protected, it is accessible within its own class as well as the class immediately derived from it.

## Visibility of Inherited Members

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

## C++ Multilevel Inheritance

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





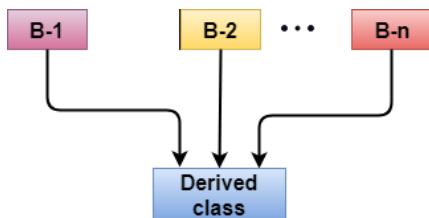
## C++ Multi Level Inheritance Example

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

Let's see the example of multi level inheritance in C++.

## C++ Multiple Inheritance

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



**Syntax of the Derived class:**

Let's see a simple example of multiple inheritance.

## Ambiguity Resolution in Inheritance

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

Let's understand this through an example:

An ambiguity can also occur in single inheritance.

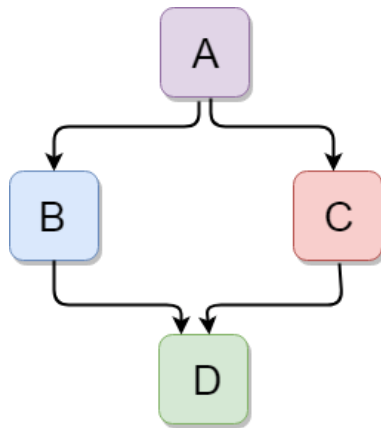
Consider the following situation:

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

```
1. int main()
2. {
3.     B b;
4.     b.display();           // Calling the display() function of B class.
5.     b.B :: display();      // Calling the display() function defined in B class.
6. }
```

## C++ Hybrid Inheritance

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



Let's see a simple example:

```
1. #include <iostream>
2. using namespace std;
3. class A
4. {
5.     protected:
6.         int a;
7.     public:
```

```
8.  void get_a()
9.  {
10.     std::cout << "Enter the value of 'a' : " << std::endl;
11.     cin>>a;
12. }
13.};
14.
15. class B : public A
16. {
17.     protected:
18.     int b;
19.     public:
20.     void get_b()
21.     {
22.         std::cout << "Enter the value of 'b' : " << std::endl;
23.         cin>>b;
24.     }
25.};
26. class C
27. {
28.     protected:
29.     int c;
30.     public:
31.     void get_c()
32.     {
33.         std::cout << "Enter the value of c is : " << std::endl;
34.         cin>>c;
35.     }
36.};
37.
38. class D : public B, public C
39. {
40.     protected:
41.     int d;
```

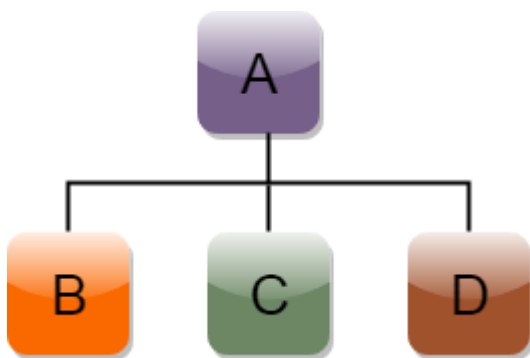
```
42. public:
43. void mul()
44. {
45.     get_a();
46.     get_b();
47.     get_c();
48.     std::cout << "Multiplication of a,b,c is : " << a*b*c << std::endl;
49. }
50. };
51. int main()
52. {
53.     D d;
54.     d.mul();
55.     return 0;
56. }
```

Output:

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

## C++ Hierarchical Inheritance

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



### Syntax of Hierarchical inheritance:

```
1. class A
2. {
3.     // body of the class A.
4. }
5. class B : public A
6. {
7.     // body of class B.
8. }
9. class C : public A
10. {
11.     // body of class C.
12. }
13. class D : public A
14. {
15.     // body of class D.
16. }
```

Let's see a simple example:

```
1. #include <iostream>
2. using namespace std;
3. class Shape           // Declaration of base class.
4. {
5.     public:
6.     int a;
7.     int b;
8.     void get_data(int n,int m)
9.     {
10.         a= n;
11.         b = m;
12.     }
13. };
14. class Rectangle : public Shape // inheriting Shape class
```

```

15. {
16.     public:
17.     int rect_area()
18.     {
19.         int result = a*b;
20.         return result;
21.     }
22. };
23. class Triangle : public Shape // inheriting Shape class
24. {
25.     public:
26.     int triangle_area()
27.     {
28.         float result = 0.5*a*b;
29.         return result;
30.     }
31. };
32. int main()
33. {
34.     Rectangle r;
35.     Triangle t;
36.     int length,breadth,base,height;
37.     std::cout << "Enter the length and breadth of a rectangle: " << std::endl;
38.     cin>>length>>breadth;
39.     r.get_data(length,breadth);
40.     int m = r.rect_area();
41.     std::cout << "Area of the rectangle is : " <<m<< std::endl;
42.     std::cout << "Enter the base and height of the triangle: " << std::endl;
43.     cin>>base>>height;
44.     t.get_data(base,height);
45.     float n = t.triangle_area();
46.     std::cout <<"Area of the triangle is : " << n<<std::endl;
47.     return 0;
48. }

```

Output:

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

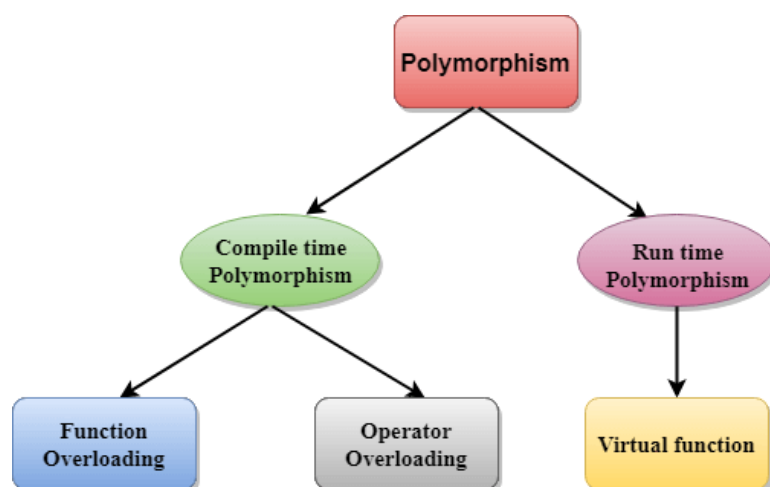
## C++ Polymorphism

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

### Real Life Example Of Polymorphism

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

**There are two types of polymorphism in C++:**



- **Compile time polymorphism:** The overloaded functions are invoked by matching the type and number of arguments. This information is available at the compile time and, therefore, compiler selects the appropriate function at the compile time. It is achieved by function overloading and operator overloading

which is also known as static binding or early binding. Now, let's consider the case where function name and prototype is same.

```
1.  class A                                // base class declaration.
2.  {
3.      int a;
4.      public:
5.      void display()
6.      {
7.          cout<< "Class A ";
8.      }
9.  };
10. class B : public A                     // derived class declaration.
11. {
12.     int b;
13.     public:
14.     void display()
15.     {
16.         cout<<"Class B";
17.     }
18.};
```

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

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

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

Compile time polymorphism	Run time polymorphism
---------------------------	-----------------------



The function to be invoked is known at the compile time.	The function to be invoked is known at the run time.
It is also known as overloading, early binding and static binding.	It is also known as overriding, Dynamic binding and late binding.
Overloading is a compile time polymorphism where more than one method is having the same name but with the different number of parameters or the type of the parameters.	Overriding is a run time polymorphism where more than one method is having the same name, number of parameters and the type of the parameters.
It is achieved by function overloading and operator overloading.	It is achieved by virtual functions and pointers.
It provides fast execution as it is known at the compile time.	It provides slow execution as it is known at the run time.
It is less flexible as mainly all the things execute at the compile time.	It is more flexible as all the things execute at the run time.

## C++ Runtime Polymorphism Example

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

// an example without the virtual keyword.

```

1. #include <iostream>
2. using namespace std;
3. class Animal {
4.     public:
5.     void eat(){
6.         cout<<"Eating...";
7.     }
8. };
9. class Dog: public Animal
10. {
11.     public:

```

```

12. void eat()
13. {      cout<<"Eating bread...";
14. }
15. };
16. int main(void) {
17.   Dog d = Dog();
18.   d.eat();
19.   return 0;
20. }

```

### Output:

```
Eating bread...
```

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

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

// an example with virtual keyword.

```

1. #include <iostream>
2. using namespace std;
3. class Shape {                                // base class
4.   public:
5.   virtual void draw(){                       // virtual function
6.     cout<<"drawing..."<<endl;
7.   }
8. };
9. class Rectangle: public Shape                // inheriting Shape class.
10. {
11.   public:
12.   void draw()
13.   {
14.     cout<<"drawing rectangle..."<<endl;

```

```

15.  }
16. };
17. class Circle: public Shape           // inheriting Shape class.
18.
19. {
20. public:
21. void draw()
22. {
23.     cout<<"drawing circle..."<<endl;
24. }
25. };
26. int main(void) {
27.     Shape *s;           // base class pointer.
28.     Shape sh;           // base class object.
29.     Rectangle rec;
30.     Circle cir;
31.     s=&sh;
32.     s->draw();
33.     s=&rec;
34.     s->draw();
35.     s=?
36.     s->draw();
37. }

```

### Output:

```

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

```

## Runtime Polymorphism with Data Members

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

```

1. #include <iostream>

```

```
2. using namespace std;
3. class Animal {                                // base class declaration.
4.     public:
5.         string color = "Black";
6. };
7. class Dog: public Animal                       // inheriting Animal class.
8. {
9.     public:
10.        string color = "Grey";
11. };
12. int main(void) {
13.     Animal d= Dog();
14.     cout<<d.color;
15. }
```

### Output:

```
Black
```

## C++ Overloading (Function and Operator)

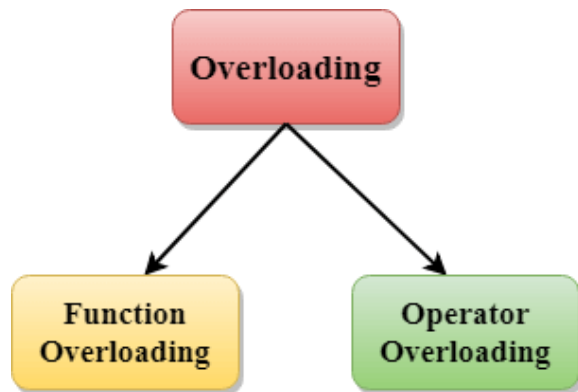
If we create two or more members having the same name but different in number or type of parameter, it is known as C++ overloading. In C++, we can overload:

- methods,
- constructors, and
- indexed properties

It is because these members have parameters only.

### Types of overloading in C++ are:

- Function overloading
- Operator overloading



## C++ Function Overloading

Function Overloading is defined as the process of having two or more function with the same name, but different in parameters is known as function overloading in C++.

The **advantage** of Function overloading is that it increases the readability of the program because you don't need to use different names for the same action.

## C++ Function Overloading Example

Let's see the simple example of function overloading where we are changing number of arguments of add() method.

// program of function overloading when number of arguments vary.

```
1. #include <iostream>
2. using namespace std;
3. class Cal {
4.     public:
5.     static int add(int a,int b){
6.         return a + b;
7.     }
8.     static int add(int a, int b, int c)
9.     {
10.         return a + b + c;
11.     }
12. };
13. int main(void) {
```

```

14.   Cal C;                                //   class object declaration.
15.   cout<<C.add(10, 20)<<endl;
16.   cout<<C.add(12, 20, 23);
17.   return 0;
18. }

```

### Output:

```

30
55

```

Let's see the simple example when the type of the arguments vary.

// Program of function overloading with different types of arguments.

```

1.  #include<iostream>
2.  using namespace std;
3.  int mul(int,int);
4.  float mul(float,int);
5.
6.
7.  int mul(int a,int b)
8.  {
9.      return a*b;
10. }
11. float mul(double x, int y)
12. {
13.     return x*y;
14. }
15. int main()
16. {
17.     int r1 = mul(6,7);
18.     float r2 = mul(0.2,3);
19.     std::cout << "r1 is : " <<r1<< std::endl;
20.     std::cout <<"r2 is : " <<r2<< std::endl;
21.     return 0;
22. }

```

**Output:**

```
r1 is : 42  
r2 is : 0.6
```

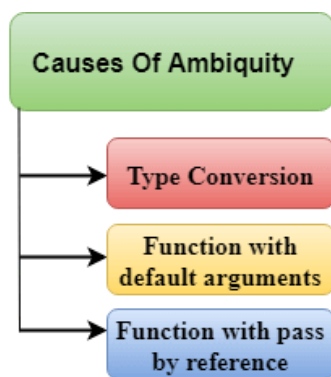
## Function Overloading and Ambiguity

When the compiler is unable to decide which function is to be invoked among the overloaded function, this situation is known as **function overloading**.

When the compiler shows the ambiguity error, the compiler does not run the program.

### Causes of Function Overloading:

- Type Conversion.
- Function with default arguments.
- Function with pass by reference.



- Type Conversion:

Let's see a simple example.

```
1. #include<iostream>
2. using namespace std;
3. void fun(int);
4. void fun(float);
5. void fun(int i)
6. {
7.     std::cout << "Value of i is : " <<i<< std::endl;
8. }
9. void fun(float j)
10. {
11.     std::cout << "Value of j is : " <<j<< std::endl;
12. }
```



```

13. int main()
14. {
15.     fun(12);
16.     fun(1.2);
17.     return 0;
18. }

```

The above example shows an error "**call of overloaded 'fun(double)' is ambiguous**".  
The fun(10) will call the first function.

- Function with Default Arguments

**Let's see a simple example.**

```

1. #include<iostream>
2. using namespace std;
3. void fun(int);
4. void fun(int,int);
5. void fun(int i)
6. {
7.     std::cout << "Value of i is : " <<i<< std::endl;
8. }
9. void fun(int a,int b=9)
10. {
11.     std::cout << "Value of a is : " <<a<< std::endl;
12.     std::cout << "Value of b is : " <<b<< std::endl;
13. }
14. int main()
15. {
16.     fun(12);
17.
18.     return 0;
19. }

```

The above example shows an error "call of overloaded 'fun(int)' is ambiguous".  
The fun(int a, int b=9) can be called in two ways: first is by calling the function with one

argument, i.e., fun(12) and another way is calling the function with two arguments, i.e., fun(4,5). The fun(int i) function is invoked with one argument.

Therefore, the compiler could not be able to select among fun(int i) and fun(int a,int b=9).

- Function with pass by reference

Let's see a simple example.

```
1. #include <iostream>
2. using namespace std;
3. void fun(int);
4. void fun(int &);
5. int main()
6. {
7.     int a=10;
8.     fun(a); // error, which f()?
9.     return 0;
10.}
11. void fun(int x)
12. {
13.     std::cout << "Value of x is : " <<x<< std::endl;
14.}
15. void fun(int &b)
16. {
17.     std::cout << "Value of b is : " <<b<< std::endl;
18.}
```

The above example shows an error "**call of overloaded 'fun(int&)' is ambiguous**". The first function takes one integer argument and the second function takes a reference parameter as an argument. In this case, the compiler does not know which function is needed by the user as there is no syntactical difference between the fun(int) and fun(int &).

## C++ Operators Overloading

Operator overloading is a compile-time polymorphism in which the operator is overloaded to provide the special meaning to the user-defined data type. Operator overloading is used to overload or redefines most of the operators available in C++. It is used to perform the operation on the user-defined data type. For example, C++ provides the ability to add the variables of the user-defined data type that is applied to the built-in data types.

The advantage of Operators overloading is to perform different operations on the same operand.

**Operator that cannot be overloaded are as follows:**

- Scope operator (::)
- Sizeof
- member selector(.)
- member pointer selector(\*)
- ternary operator(?:)

## Syntax of Operator Overloading

1. return\_type class\_name :: operator op(argument\_list)
2. {
3.     // body of the function.
4. }

Where the **return type** is the type of value returned by the function.

**class\_name** is the name of the class.

**operator op** is an operator function where op is the operator being overloaded, and the operator is the keyword.

## Rules for Operator Overloading

- Existing operators can only be overloaded, but the new operators cannot be overloaded.
- The overloaded operator contains atleast one operand of the user-defined data type.

- We cannot use friend function to overload certain operators. However, the member function can be used to overload those operators.
- When unary operators are overloaded through a member function take no explicit arguments, but, if they are overloaded by a friend function, takes one argument.
- When binary operators are overloaded through a member function takes one explicit argument, and if they are overloaded through a friend function takes two explicit arguments.

## C++ Operators Overloading Example

Let's see the simple example of operator overloading in C++. In this example, void operator ++ () operator function is defined (inside Test class).

// program to overload the unary operator ++.

```
1. #include <iostream>
2. using namespace std;
3. class Test
4. {
5.     private:
6.         int num;
7.     public:
8.         Test(): num(8){}
9.         void operator ++()    {
10.             num = num+2;
11.         }
12.         void Print() {
13.             cout<<"The Count is: "<<num;
14.         }
15. };
16. int main()
17. {
18.     Test tt;
19.     ++tt; // calling of a function "void operator ++()"
```

```
20. tt.Print();
21. return 0;
22. }
```

### Output:

```
The Count is: 10
```

## C++ Interfaces

**Interfaces** play an important role in designing clean and well-organized code structures in C++. In C++, an interface is a conceptual construct that specifies a set of methods that must be implemented by any class that claims to conform to the Interface. It acts as a blueprint that enforces consistent behavior among its implementers while shielding the details of the actual implementation. By doing so, C++ interfaces enhance code modularity, maintainability, and reusability, aligning perfectly with the principles of object-oriented programming.

### Interface as a Concept

Although C++ lacks a defined **interface keyword**, the idea of an interface is accomplished through **abstract classes** and pure **virtual functions**. An abstract class has one or more pure virtual functions - functions specified in the base class but not implemented. These pure virtual functions serve as the methods of the Interface, setting up the requirements for any class that derives from the abstract base.

```
1. class Interface {
2. public:
3. virtual void method1() = 0; // Pure virtual function
4. virtual void method2() = 0; // Pure virtual function
5. // ... more pure virtual functions
6. };
```

### Implementing the Interface

**Classes** that implement the interface must override all pure virtual functions declared in the **abstract base**. These classes provide the necessary implementations for the interface methods, adhering to the prescribed contract.

## C++

```
1. class ConcreteClass: public Interface {  
2. public:  
3. void method1() override {  
4. // Implementation for method1  
5. }  
6. void method2() override {  
7. // Implementation for method2  
8. }  
9. };
```

### Example 1:

```
1. #include <iostream>  
2.  
3. // Define the Shape interface  
4. class Shape {  
5. public:  
6. virtual double getArea() const = 0; // Pure virtual function for area  
7. virtual double getPerimeter() const = 0; // Pure virtual function for perimeter  
8. };  
9.  
10. // Implement the Interface for a Circle  
11. class Circle: public Shape {  
12. private:  
13. double radius;  
14.  
15. public:  
16. Circle(double r) : radius(r) {}  
17.  
18. double getArea() const override {  
19. return 3.14159265359 * radius * radius;  
20. }  
21.
```

```

22. double getPerimeter() const override {
23. return 2 * 3.14159265359 * radius;
24. }
25. };
26.
27. // Implement the Interface for a Rectangle
28. class Rectangle: public Shape {
29. private:
30. double length;
31. double width;
32.
33. public:
34. Rectangle(double l, double w) : length(l), width(w) {}
35.
36. double getArea() const override {
37. return length * width;
38. }
39.
40. double getPerimeter() const override {
41. return 2 * (length + width);
42. }
43. };
44.
45. int main() {
46. // Create instances of Circle and Rectangle
47. Circle circle(5.0);
48. Rectangle rectangle(4.0, 6.0);
49.
50. // Calculate and display the area and perimeter of each shape
51. std::cout << "Circle Area: " << circle.getArea() << ", Perimeter: " << circle.getPerimeter()
    << std::endl;
52. std::cout << "Rectangle Area: " << rectangle.getArea() << ", Perimeter: " << rectangle.g
    etPerimeter() << std::endl;
53.

```

```
54. return 0;
55. }
```

### Output:

```
Circle Area: 78.5398, Perimeter: 31.4159
Rectangle Area: 24, Perimeter: 20
```

## Pure Virtual Function:

A **pure virtual function** is a function in C++ specified in a base class that does not have an implementation, signified by "**= 0**" in its declaration. It serves as a contract that derived classes must implement.

### C++

1. **virtual void** myFunction() = 0;
  - "**virtual**": It indicates that this function is virtual, enabling polymorphism.
  - "**void**": Specifies the return type.
  - "**myFunction**": Name of the function.
  - "**= 0**": Marks it as pure virtual, meaning it has no implementation in the base class and must be overridden in derived classes.

# C++ Functions

A function is a group of statements that together perform a task.

Every C++ program has at least one function, which is **main()**, and all the most trivial programs can define additional functions.

You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division usually is such that each function performs a specific task.

A function **declaration** tells the compiler about a function's name, return type, and parameters. A function **definition** provides the actual body of the function.

The C++ standard library provides numerous built-in functions that your program can call. For example, function **strcat()** to concatenate two strings,



function **memcpy()** to copy one memory location to another location and many more functions.

A function is known with various names like a method or a sub-routine or a procedure etc.

## Defining a Function

The general form of a C++ function definition is as follows –

```
return_type function_name( parameter list ) {  
    body of the function  
}
```

A C++ function definition consists of a function header and a function body. Here are all the parts of a function –

- **Return Type** – A function may return a value. The **return\_type** is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the **return\_type** is the keyword **void**.
- **Function Name** – This is the actual name of the function. The function name and the parameter list together constitute the function signature.
- **Parameters** – A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.
- **Function Body** – The function body contains a collection of statements that define what the function does.

## Example

Following is the source code for a function called **max()**. This function takes two parameters **num1** and **num2** and return the biggest of both –

```
// function returning the max between two numbers
```

```
int max(int num1, int num2) {  
    // local variable declaration  
    int result;  
  
    if (num1 > num2)  
        result = num1;  
    else  
        result = num2;  
  
    return result;  
}
```

## Function Declarations

A function **declaration** tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately.

A function declaration has the following parts –

```
return_type function_name( parameter list );
```

For the above defined function `max()`, following is the function declaration –

```
int max(int num1, int num2);
```

Parameter names are not important in function declaration only their type is required, so following is also valid declaration –

```
int max(int, int);
```

Function declaration is required when you define a function in one source file and you call that function in another file. In such case, you should declare the function at the top of the file calling the function.

## Calling a Function

While creating a C++ function, you give a definition of what the function has to do. To use a function, you will have to call or invoke that function.

When a program calls a function, program control is transferred to the called function. A called function performs defined task and when it's return statement is executed or when its function-ending closing brace is reached, it returns program control back to the main program.

To call a function, you simply need to pass the required parameters along with function name, and if function returns a value, then you can store returned value. For example –

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

// function declaration
int max(int num1, int num2);

int main () {
    // local variable declaration:
    int a = 100;
    int b = 200;
    int ret;

    // calling a function to get max value.
    ret = max(a, b);
    cout << "Max value is : " << ret << endl;

    return 0;
}

// function returning the max between two numbers
int max(int num1, int num2) {
    // local variable declaration
    int result;

    if (num1 > num2)
        result = num1;
    else
        result = num2;

    return result;
}
```

I kept max() function along with main() function and compiled the source code. While running final executable, it would produce the following result –

Max value is : 200

## Function Arguments

If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the **formal parameters** of the function.

The formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.

While calling a function, there are two ways that arguments can be passed to a function –

## POINTER IN C++

In C++, pointers are variables that store memory addresses. Rather than holding data values directly, a pointer holds the address of another variable where the data is stored. Pointers are a powerful feature in C++ that allow for more flexible and efficient handling of data and memory management.

### Key Concepts of Pointers

#### 1. Declaration and Initialization:

- **Declaration:** A pointer is declared by specifying the type of data it will point to, followed by an asterisk \*. For example, `int *ptr;` declares a pointer to an integer.
- **Initialization:** You typically initialize a pointer with the address of a variable using the address-of operator `&`. For example:

```
int a = 10;
int *ptr = &a;    // ptr now holds the address of variable a
```

#### 2. Dereferencing:

- Dereferencing a pointer means accessing the value at the address the pointer is pointing to. This is done using the asterisk \*. For example:

```
int value = *ptr; // Gets the value stored at the address held by ptr
```

#### 3. Null Pointer:

- A null pointer is a pointer that does not point to any valid memory location. It is often used to indicate that the pointer is not currently assigned to any variable. It can be initialized using `nullptr` in C++11 and later, or `NULL` in older C++ standards:

```
int *ptr = nullptr;    // C++11 and later
```

#### 4. Pointer Arithmetic:

- You can perform arithmetic operations on pointers, such as incrementing or decrementing them. Pointer arithmetic is based on the size of the data type the pointer points to. For example:

```
int arr[5] = {1, 2, 3, 4, 5};
int *ptr = arr; // Points to the first element of the array
ptr++; // Moves the pointer to the next integer in the array
```

#### 5. Pointers and Arrays:

- In C++, arrays and pointers are closely related. An array name can be used as a pointer to its first element. For example:

```
int arr[5] = {1, 2, 3, 4, 5};
int *ptr = arr; // Equivalent to int *ptr = &arr[0];
```

#### 6. Dynamic Memory Allocation:

- Pointers are essential for dynamic memory management. The `new` operator is used to allocate memory dynamically on the heap, and the `delete` operator is used to free that memory:

```
int *ptr = new int; // Allocates memory for a single integer
*ptr = 10;
delete ptr; // Deallocates memory

int *arr = new int[5]; // Allocates memory for an array of 5
integers
delete[] arr; // Deallocates memory for the array
```

#### 7. Pointer to Pointer:

- C++ allows pointers to point to other pointers, creating a multi-level pointer. For example:




```
int a = 10;
int *ptr = &a;
int **ptrToPtr = &ptr; // Pointer to pointer
```

#### 8. Function Pointers:

- 9. Pointers can also be used to point to functions, allowing for dynamic function calls and callback mechanisms:

```
10. void func() {
11.     std::cout << "Hello, world!" << std::endl;
12. }
13. void (*funcPtr)() = func; // Pointer to function
14. funcPtr(); // Calls func
```

CODE

main.cpp		   Share	Run	Output
<pre>1 // Online C++ compiler to run C++ program online 2 #include &lt;iostream&gt; 3 4 void showPointerDetails(int *ptr) { 5     std::cout &lt;&lt; "Value: " &lt;&lt; *ptr &lt;&lt; std::endl; 6     std::cout &lt;&lt; "Address: " &lt;&lt; ptr &lt;&lt; std::endl; 7 } 8 9 int main() { 10     int a = 10; 11     int *ptr = &amp;a; 12 13     std::cout &lt;&lt; "Initial value of a: " &lt;&lt; a &lt;&lt; std::endl; 14     std::cout &lt;&lt; "Pointer ptr points to address: " &lt;&lt; ptr &lt;&lt; std::endl; 15     std::cout &lt;&lt; "Value at address pointed by ptr: " &lt;&lt; *ptr &lt;&lt; std::endl; 16 17     *ptr = 20; // Modify the value of a through the pointer 18     std::cout &lt;&lt; "New value of a: " &lt;&lt; a &lt;&lt; std::endl; 19 20     // Function pointer example 21     void (*funcPtr)() = []() { 22         std::cout &lt;&lt; "Function pointer called!" &lt;&lt; std::endl; 23     }; 24     funcPtr(); 25 26     showPointerDetails(ptr); 27 28     return 0; 29 } 30</pre>		<pre>/tmp/PhkLdkpCTK.o Initial value of a: 10 Pointer ptr points to address: 0x7fff8a88a90c Value at address pointed by ptr: 10 New value of a: 20 Function pointer called! Value: 20 Address: 0x7fff8a88a90c  === Code Execution Successful ===</pre>		