C++ notes

Simple Program

```
#include <iostream>

using namespace std;


// main() is where program execution begins.

int main() {
   cout << "Hello World"; // prints Hello World

   return 0;
}
```

New Line Add

```
#include <iostream>
using namespace std;
int main() {
   cout << "Hello World"<<endl;
   cout << "welcome "<<endl;
   cout << "bye"<<endl;
   return 0;
}
```

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

int main() {
   cout << "Size of char : " << sizeof(char) << endl;
   cout << "Size of int : " << sizeof(int) << endl;
   cout << "Size of short int : " << sizeof(short int) << endl;
   cout << "Size of long int : " << sizeof(long int) << endl;
   cout << "Size of float : " << sizeof(float) << endl;
   cout << "Size of double : " << sizeof(double) << endl;
   cout << "Size of wchar_t : " << sizeof(wchar_t) << endl;

   return 0;
}
```

Create const variables

A `const` variable must be initialized during declaration and cannot be changed later.

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

int main() {

    // initialize a const PI
    const double PI = 3.14;

    int radius = 4;

    // use the const in a calculation
    double area = PI * radius * radius;

    cout << "Area of circle with radius " << radius
<< " is: " << area;

    return 0;
}
```

Input/Output

```
#include <iostream>

using namespace std;


int main() {

    int num;

    cout << "Enter an integer: ";

    cin >> num;   // Taking input

    cout << "The number is: " << num;//output

    return 0;

}
```

CHAR

```
#include <iostream>
using namespace std;
int main() {
    char a;
    int num;

    cout << "Enter a character and an integer: ";
    cin >> a >> num;

    cout << "Character: " << a << endl;
    cout << "Number: " << num;

    return 0;
}
```

STRING

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

int main() {
    string num1;


    cout << "Enter first number: ";
    cin >> num1;

    // Display the result
    cout << "ans " << num1 <<endl;

    return 0;
}
```

Operators in C++

- An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. C++ is rich in built-in operators and provide the following types of operators −Aritmetic Operators Relational Operators Logical Operators Bitwise Operators Assignment Operators Misc Operators This chapter will examine the arithmetic, relational, logical, bitwise, assignment and other operators one by one.

Arithmetic Operators

There are following arithmetic

operators supported by C++

language − Assume variable A holds

10 and variable B holds 20, then −

Show Examples

Operator	Description	Example
+	Adds two operands	A + B will give 30

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

--	**Decrement operator**, decreases integer value by one	A-- will give 9

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

Relational Operators

There are following relational operators supported by C++ language Assume variable A holds 10 and variable B holds 20, then −

Show Examples

Logical Operators

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

Show Examples

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

Bitwise Operators

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

p	q	p & q	p | q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0

1	0	0	1	1

Assume if A = 60; and B = 13; now in binary format

they will be as follows − A = 0011 1100

B = 0000 1101

A&B = 0000 1100

A|B = 0011 1101

A^B = 0011 0001

~A = 1100 0011

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

Show Examples

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

	number of bits specified by the right operand.	

Assignment Operators

There are following assignment operators

supported by C++ language − Show Examples

Operator	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand.	C = A + B will assign value of A + B into C
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand.	C += A is equivalent to C = C + A

-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand.	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand.	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand.	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand.	C %= A is equivalent to C = C % A
<<=	Left shift AND assignment operator.	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator.	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator.	C &= 2 is same as C = C & 2
^=	Bitwise exclusive OR and assignment operator.	C ^= 2 is same as C = C ^ 2
|=	Bitwise inclusive OR and	C |= 2 is same as C =

	assignment operator.	C | 2

Misc Operators

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

Sr. No	Operator & Description
1	**sizeof** **sizeof operator** returns the size of a variable. For example, sizeof(a), where'a' is integer, and will return 4.
2	**Condition ? X : Y** **Conditional operator (?)**. If Condition is true then it returns value of X otherwise returns value of Y.
3	**,**

	Comma operator causes a sequence of operations to be performed. The value of the entire comma expression is the value of the last expression of the comma-separated list.
4	**. (dot) and -> (arrow)** **Member operators** are used to reference individual members of classes, structures, and unions.
5	**Cast** **Casting operators** convert one data type to another. For example, int(2.2000) would return 2.
6	**&** **Pointer operator &** returns the address of a variable. For example &a; will give actual address of the variable.
7	***** **Pointer operator *** is pointer to a variable. For example *var; will pointer to a variable var.

Category	Operator	Associativity
Postfix	() [] -> . ++ - -	Left to right
Unary	+ - ! ~ ++ - - (type)* & sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >>	Left to right

Operators Precedence in C++

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator −

For example x = 7 + 3 * 2; here, x is assigned 13, not 20 because operator * has higher precedence than +, so it first gets multiplied with 3*2 and then adds into 7.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

Show Examples

<table>
<tr><td>Relational</td><td>< <= > >=</td><td>Left to right</td></tr>
<tr><td>Equality</td><td>== !=</td><td>Left to right</td></tr>
<tr><td>Bitwise AND</td><td>&</td><td>Left to right</td></tr>
<tr><td>Bitwise XOR</td><td>^</td><td>Left to right</td></tr>
<tr><td>Bitwise OR</td><td>|</td><td>Left to right</td></tr>
<tr><td>Logical AND</td><td>&&</td><td>Left to right</td></tr>
<tr><td>Logical OR</td><td>||</td><td>Left to right</td></tr>
<tr><td>Conditional</td><td>?:</td><td>Right to left</td></tr>
<tr><td>Assignment</td><td>= += -= *= /= %=>>= <<= &= ^= |=</td><td>Right to left</td></tr>
<tr><td>Comma</td><td>,</td><td>Left to right</td></tr>
</table>

Operator

Arithmetic Operators

```
#include <iostream>
using namespace std;
int main() {
   int a, b;
   cout<<"enter no a=";
   cin>>a;
   cout<<"enter no b=";
   cin>>b;
   // 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;
}
```

// Working of increment and decrement operators

```
#include <iostream>
using namespace std;
int main() {
   int a, b ;
     cout<<"enter no a=";
   cin>>a;
   cout<<"enter no b=";
```

```
    cin>>b;
    // incrementing a by 1 and storing the result in
result_a
    a = ++a;
    cout << "result_a = " << a << endl;
    // decrementing b by 1 and storing the result in
result_b
    b = --b;
    cout << "result_b = " << b << endl;
    return 0;
}
```

Assignment Operators

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

int main() {
    int a, b;
        cout<<"enter no a=";
    cin>>a;
    cout<<"enter no b=";
    cin>>b;
    // assigning the sum of a and b to a
    a += b;  // a = a +b//*,-,/
    cout << "a = " << a << endl;

    return 0;
}
```

Relational Operators

```
#include <iostream>
using namespace std;
int main() {
    int a, b,result;
        cout<<"enter no a=";
    cin>>a;
    cout<<"enter no b=";
    cin>>b;
    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;
}
```

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;
}
```

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

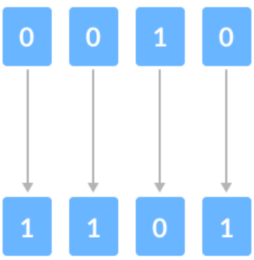
Output

```
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.

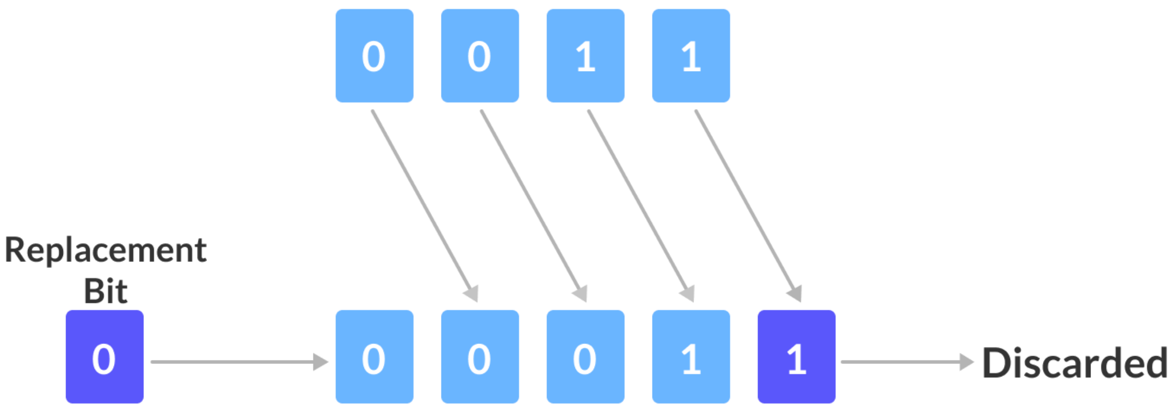
C++ Shift Operators

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.

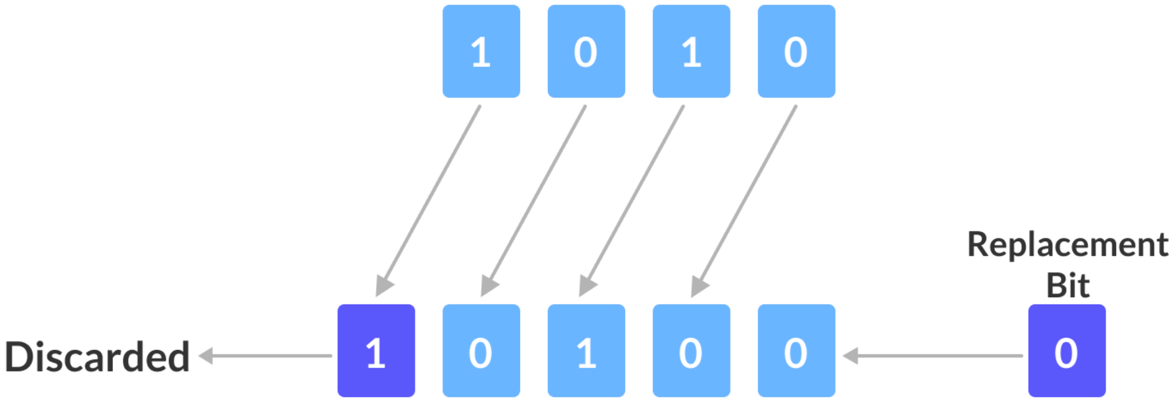


As we can see from the image above, we have a **4-bit number**. When we perform a **one-bit** right shift operation on it, each individual bit is shifted to the right by 1 bit.

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 `<<`.



One bit Left Shift

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.

Control Statament

IF STAEMNT

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

  int a,b;

  cout<<"etnter no a=";
  cin>>a;
  cout<<"etnter no b=";
  cin>>b;
                  if (a > b) {
    cout << " a is greater than b" << endl;
}
return 0;
}
```

_

If else

```
#include<iostream>
using namespace std;
int main() {
  int a,b;
  cout<<"etnter no a=";
  cin>>a;
  cout<<"etnter no b=";
  cin>>b;
                  if (a >= b) {
    cout << "You are eligible to vote";
}
else {
      cout << "You are not eligible to vote";
  }
return 0;
}
```

if else if

```
#include <iostream>
using namespace std;
int main() {
  int marks;
  cout<<"etnter no marks=";
  cin>>marks;

  if(marks >= 80)
    cout << "Amazing Grade : A";
  else if(marks >= 70)
    cout << "Grade B";
  else if(marks >= 50)
    cout << "Grade C";
  else if (marks >= 33)
    cout << "Grade D";
  else
    cout << "Failed !";
  return 0;
}
```

Short Hand If Else

conditional operator

```
#include <iostream>
using namespace std;
int main() {
  int age = 30;
  string message = (age >=18) ? "You are eligible
to vote" : "You are not eligible to vote";
  cout << message;
  return 0;
}
```

LOOPS

While loop

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

int main() {
     int count;
    cout<<"enter number=";
  cin>>count;


    while (count <= 5) {
        cout << "Count: " << count << endl;
        count++;
    }

    cout << "Loop finished." << endl;

    return 0;
}
```

Do while loop

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

int main() {
    int count;
    cout<<"enter number=";
  cin>>count;


     do{
        cout << "Count: " << count << endl;
        count++;
    }while (count <= 5);

    cout << "Loop finished." << endl;

    return 0;
}
```

For Loop

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

int main() {
                    int i;
                    cout<<"enter no =";
  cin>>i;

    for ( i; i <= 5; ++i) {
        cout << "Iteration: " << i <<endl;
    }

    return 0;
}
```

CONTINUE

```
#include <iostream>
using namespace std;
int main() {
    int i;
    cout<<"enter no =";
  cin>>i;
    while (i < 10) {
        i++;
        // Skip the iteration if i is 2
        if (i == 5) {
            continue;  // Skip to the next iteration
        }
        cout << "Current number: " << i <<endl;
    }

    return 0;
}
```

BREAK

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

int main() {
                    int i;
  cout<<"enter no =";
  cin>>i;
    for (int i = 1; i <= 5; ++i) {
        cout << "Current number: " << i <<endl;
        // Break the loop if i is 3
        if (i == 3) {
            break;  // Exit the loop
        }
    }

    return 0;
}
```

Goto Statement

Transfers Control To The Labeled Statement. Though It Is Not Advised To Use Goto Statement In Your Program

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

      cout<<"Enter your age:\n";
      int age;
      cin>>age;
      if (age < 18){
                  cout<<"You are not eligible to vote!\n";
            goto ineligible;
      }
      else
      {
            cout<<"You are eligible to vote!";
      }

                  return 0;
}
```

C++ switch

The ***switch statement*** in C++ is a potent ***control structure*** that enables you to run several code segments based on the result of an expression. It offers a sophisticated and effective substitute for utilizing a succession of ***if-else-if statements*** when you have to make a decision between several possibilities.

```
#include <iostream>
using namespace std;
int main () {
      int num;
      cout<<"Enter a number to check:";
      cin>>num;
         switch (num)
        {
            case 10: cout<<"It is 10";
                       break;
            case 20: cout<<"It is 20";
                       break;
            case 30: cout<<"It is 30";
                       break;
            default: cout<<"Not 10, 20 or 30"; break;
        }
                 return 0;
    }
```

Function

Calling a Function

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

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

int main() {

    // calling the function
    greet();

    return 0;
}
```

Function with Parameters Call by reference in C+

```
#include <stdio.h>

void sum(int a, int b)//create function
{
    int total;

    total=a+b;

    printf("Total : %d",total);
}

int main()
{
    int num1,num2;

    printf("Enter First Number : ");
    scanf("%d",&num1);
    printf("Enter Second Number : ");
```

```
    scanf("%d",&num2);

    sum(num1, num2);

    return 0;
}
```

Swap values function

```
#include <iostream>
using namespace std;
// Function prototype
void swapValues(int &a, int &b);
int main() {
    int num1 = 5, num2 = 10;
    cout << "Before swapping:" << endl;
    cout << "num1 = " << num1 << ", num2 = " << num2 << endl;
    // Call the swapValues function to swap num1 and num2
    swapValues(num1, num2);
    cout << "After swapping:" << endl;
    cout << "num1 = " << num1 << ", num2 = " << num2 << endl;
    return 0;
}
// Function definition to swap values
void swapValues(int &a, int &b) {
    int temp = a;  // Store the value of a in a temporary variable
    a = b;         // Assign the value of b to a
    b = temp;      // Assign the value of the temporary variable to b
}
```

Recursion

Recursion in C++ involves a function calling itself directly or indirectly to solve a problem. Recursion can be a powerful technique in programming, especially for solving problems that can be broken down into smaller, similar sub-problems. Let's look at a simple example of recursion in C++.

Example: Factorial Calculation Using Recursion

Here's a program that calculates the factorial of a number using recursion:

```
#include <iostream>
using namespace std;
// Function prototype
int factorial(int n);
int main() {
    int number;
    cout << "Enter a non-negative integer: ";
    cin >> number;
    if (number < 0) {
        cout << "Factorial is not defined for negative numbers." << endl;
    } else {
        int result = factorial(number);
        cout << "Factorial of " << number << " is: " << result << endl;
    }
    return 0;
}
// Recursive function to calculate factorial
int factorial(int n) {
    // Base case: factorial of 0 is 1
    if (n == 0) {
        return 1;
    }
    // Recursive case: n * factorial(n-1)
    else {
        return n * factorial(n - 1);
    }
}
```

Array

1D array

```
#include <iostream>
using namespace std;
int main() {
  int i,n,numbers[5];
  cout << "Enter 5 numbers: " << endl;
  //  store input from user to array
  for (i = 0; i < 5; ++i) {
    cin >> numbers[i];
  }
  cout << "The numbers are: ";
  //  print array elements
for (i = 0; i < 5; ++i) {
    cout<< numbers[i]<<endl;
  }
  return 0;
}
```

2D ARRAY

```
#include <iostream>
using namespace std;
int main() {
    int i, j;
    int arr[3][5];
    // Input elements into the 2D array
    for (i = 0; i < 3; ++i) {
        for (j = 0; j < 5; ++j) {
            cout << "Enter element for row " << i <<
", column " << j << ": ";
            cin >> arr[i][j];
        }
    }
    // Output the 2D array elements
    cout << "\n2D Array Elements:\n";
    for (i = 0; i < 3; ++i) {
        for (j = 0; j < 5; ++j) {
            cout << arr[i][j] << " ";
        }
        cout << "\n";
    }
    return 0;
}
```

3D ARRAY

```
#include <iostream>
using namespace std;
int main() {
    int t, i, j;
    int arr[2][3][5];
    // Input elements
    for (t = 0; t < 2; t++) {
        for (i = 0; i < 3; i++) {
            for (j = 0; j < 5; j++) {
                cout << "Enter " << (t + 1) << " table "
<< i << " row " << j << " column element: ";
                cin >> arr[t][i][j];
            }
        }
    }
    // Output elements
    for (t = 0; t < 2; t++) {
        cout << "\n" << (t + 1) << " Table Array
Elements:\n";
        for (i = 0; i < 3; i++) {
            for (j = 0; j < 5; j++) {
                cout << arr[t][i][j] << " ";
            }
            cout << endl;
        }
    }
    return 0;
}
```

There are five types of storage classes, which can be used in a C++ program

1. **Automatic**
2. **Register**
3. **Static**
4. **External**
5. **Mutable**

AUTO: **It is the default storage class for local variables declared within a block or function. Variables with the *"auto"* storage class have automatic storage duration, meaning they are created when the block or function is entered and destroyed when it's exited. They are typically stored on the stack. the *"auto" keyword* is rarely used directly because local variables are considered *"auto"* by default.**

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

void autoStorageClass() {
   cout << "Understanding the auto keyword for type deduction\n";

   // Declaring variables with auto keyword
   auto a = 32;           // 'a' will be of type int
   auto b = 3.2f;         // 'b' will be of type float
   auto c = "JavaTpoint"; // 'c' will be of type const char*
   auto d = 'G';          // 'd' will be of type char

   // Displaying the values of auto variables
   cout << a << " \n";    // Output: 32
   cout << b << " \n";    // Output: 3.2
```

```
    cout << c << " \n";    // Output: JavaTpoint
    cout << d << " \n";    // Output: G
}

int main() {
    // Example of auto keyword
    autoStorageClass();

    return 0;
}
```

The extern

Storage Class in C++ for Global Variables

In C++, the storage classes play a crucial role in managing the scope, memory allocation, and lifetime of variables. Among these, the *'extern'* storage class stands out by allowing variables to be defined in a different block and accessed across multiple files.

Properties of the extern Storage Class:

The *'extern'* storage class is characterized by specific attributes that dictate its behaviour:

Scope: Global

The *'extern'* storage class gives a variable global scope. It means that the variable can be accessed and used anywhere within the program, even outside the block where it was declared.

Default Value: Zero Initialization

By default, the *'extern'* variables are initialized to zero. It implies that if the variable is not explicitly assigned a value, it will hold a value of zero.

Memory Location: RAM

Similar to other variables, the *'extern'* variables are allocated memory in the computer's RAM (Random Access Memory). This memory space is used to store the variable's value during program execution.

Lifetime: Lasts Until Program Termination

The lifetime of an *'extern'* variable extends throughout the entire duration of the program. It is created when the program starts and retains its value until the program terminates.

Fil1.cpp

```
#include <iostream>
using namespace std;
// Function declaration
void myfun();
// Global variable
int num = 10;

int main() {
    myfun();
    return 0;
}
// Function definition
void myfun() {
    cout << num << endl;
}
```

File2

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

extern int num;     //extern global variable

int main()
{
cout<<num;

    return 0;
}
```

The static Storage Class in C++

In the world of C++, storage classes play a vital role in managing variables' scope, lifetime, and memory allocation. Among these, the *'static'* storage class stands out for its unique ability to preserve values even after they leave their scope.

Properties of the static Storage Class:

The *'static'* storage class is characterized by specific attributes that govern its behavior:

Scope: Local

The *'static'* storage class confines variables to their local scope. However, unlike other local variables, the *'static'* variables retain their value across function calls.

Default Value: Zero Initialization

The *'static'* variables are automatically initialized to zero if not explicitly assigned a value. This ensures a predictable starting point for their values.

Memory Location: RAM

Like other variables, the *'static'* variables are allocated memory in the computer's RAM (Random Access Memory). This memory space holds the variable's value throughout the program's execution.

Lifetime: Lasts Until Program Termination

The *'static'* variables persist in memory until the program concludes. Their values are preserved across function calls, making them a suitable choice for maintaining state information.

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


void myfun();

int main() {
    myfun();
    myfun();
    myfun();

    return 0;
}

void myfun() {
    static int i = 10;  // Static variable to retain its value across function calls
    cout << i <<endl;  // Use std::cout for output
    i++;  // Increment the static variable
}
```

The register Storage Class in C++

In the realm of C++, storage classes wield influence over the behavior of variables, governing aspects like *scope, memory allocation*, and *lifetime*. The *'register'* storage class, once utilized for optimizing variable access, is introduced in this article. However, it's important to note that as of C++17, the *'register'* keyword is deprecated, and modern compilers often optimize variable storage and access automatically.

Properties of the register Storage Class:

The *'register'* storage class is distinguished by specific traits that characterize its functionality:

Scope: Local

Similar to the *'auto'* variables, *'register'* variables possess local scope, confined to the block in which they are declared.

Default Value: Indeterminate (Garbage Value)

Like *'auto'* variables, *'register'* variables are not automatically initialized. Their initial values are indeterminate until explicitly assigned.

Memory Location: CPU Register or RAM

While *'register'* variables aim to reside within a CPU register for faster access, this behaviour depends on the availability of free registers. If no register is available, these variables are stored in RAM like ordinary variables.

Lifetime: Limited to Block Scope

The lifespan of *'register'* variables is bounded by the block in which they are declared. Upon exiting the block, the variables cease to exist.

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

void registerStorageClass() {
    cout << "Illustrating the register class\n";

    // Declaring a register variable
    register char b = 'G';

    // Displaying the value of the register variable 'b'
    cout << "Value of the variable 'b' declared as register: " << b;
}

int main() {
    // Demonstrating the register Storage Class
    registerStorageClass();
    return 0;
}
```

Pointer

```
#include <iostream>
using namespace std;
 // For std::cout

int main() {
    int num;
    int* ptr;

    num = 25;
    ptr = &num;

    // Use std::cout for output
    cout << "Value of Num : " << num << endl;
    cout << "Address of Num : " << &num << endl;

    cout << "Value of Ptr : " << ptr << endl;
    cout << "Address of Ptr : " << &ptr << endl;

    return 0;
}
```

Structure

```cpp
#include <iostream>
#include <string>  // For std::string

using namespace std;

struct Student
{
    int roll;
    string name;  // Use std::string instead of char array
    float cgpa;
};

int main()
{
    Student stu1;  // Use the C++ struct type
    cout << "Enter Student Roll No.: ";
    cin >> stu1.roll;
    cout << "Enter Student Name: ";
    cin.ignore();  // To clear the newline left by previous input
    getline(cin, stu1.name);  // Read the full line for name

    cout << "Enter Student CGPA: ";
    cin >> stu1.cgpa;

    cout << "\nStudent Roll No.: " << stu1.roll << endl;
    cout << "Student Name: " << stu1.name << endl;
    cout << "Student CGPA: " << stu1.cgpa << endl;

    return 0;
}
```

```
#include <iostream>
#include <string>  // For std::string
using namespace std;
struct Student
{
    int roll;
    string name;  // Use std::string instead of char
array
    float cgpa;
};
int main()
{
    Student stu[3];  // Array of Student structs
    int i;

    // Input data for each student
    for (i = 0; i < 3; i++)
    {
        cout << "\nEnter Student Roll No.: ";
        cin >> stu[i].roll;
        cout << "Enter Student Name: ";
        cin.ignore();  // Clear the newline character
left by previous input
        getline(cin, stu[i].name);  // Read the full
line for name
        cout << "Enter Student CGPA: ";
        cin >> stu[i].cgpa;
    }
    // Output data for each student
    for (i = 0; i < 3; i++)
    {
        cout << "\nStudent Roll No.: " << stu[i].roll
             << ", Name: " << stu[i].name
             << ", CGPA: " << stu[i].cgpa;
    }

    return 0;
}
```

STRING

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

int main()
{
    string name;
    cout << "Enter name:";
    getline (cin, name);
    cout << "Hello " << name;
    return 0;
}
```

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


int main() {
    string str;

    cout << "Enter Your Name: ";
    getline(cin, str);  // Safely reads a line of input including spaces

    cout << "Your Name: " << str << endl;

    return 0;
}
```

Str length

```
#include <iostream>
#include <string>
using namespace std;
int main() {
    string str;
    int len;


    cout << "Enter Your String: ";
    getline(cin, str);  // Read a full line of input
into the string

    len = str.length();  // Get the length of the
string

    cout << "String Length: " << len << endl;

    return 0;
}
```

Reverse

```
#include <iostream>
#include <algorithm> // For std::reverse
#include <string>    // For std::string

using namespace std;

int main()
{
    string str;  // Use a single string instead of an array

    cout << "Enter Your String: ";
    getline(cin, str);  // Read the entire line into 'str'

    // Reverse the string using std::reverse from <algorithm>
    reverse(str.begin(), str.end());

    // Output the reversed string
    cout << "Reversed String: " << str << endl;

    return 0;
```

Uppercase

```cpp
#include <iostream>
#include <algorithm> // For std::reverse and std::transform
#include <cctype>    // For std::toupper
#include <string>    // For std::string
using namespace std;
int main()
{
    string str;  // Use a single string instead of an array
    cout << "Enter Your String: ";
    getline(cin, str);  // Read the entire line into 'str'
    // Reverse the string using std::reverse from <algorithm>
    reverse(str.begin(), str.end());
    // Convert the string to uppercase using std::transform and std::toupper
    transform(str.begin(), str.end(), str.begin(), ::toupper);
    // Output the reversed and uppercased string
    cout << "Reversed and Uppercase String: " << str << endl;
    return 0
```

TOLOWERCASE PROGRAM IN C

```
#include <iostream>
#include <algorithm> // For std::transform
#include <cctype>    // For std::tolower
#include <string>    // For std::string

using namespace std;

int main()
{
    string str;  // Declare a string variable

    cout << "Enter Your String: ";
    getline(cin, str);  // Read the entire line into 'str'

    // Convert the string to lowercase using std::transform and std::tolower
    transform(str.begin(), str.end(), str.begin(), ::tolower);

    // Output the lowercase string
    cout << "Lowercase String: " << str << endl;

    return 0;
}
```

Copy

```cpp
#include <iostream>
#include <string>  // For std::string and std::getline

using namespace std;

int main()
{
    string str, dstr;

    cout << "Enter Your String: ";
    getline(cin, str);  // Read the entire line into 'str'

    dstr = str;  // Copy 'str' to 'dstr'

    cout << "Str value: " << str << endl;
    cout << "DStr value: " << dstr << endl;

    return 0;
}
```

Concenate Meanse Both Are Connect

```
#include <iostream>
#include <string>  // For std::string
using namespace std;
int main()
{
   string str1, str2;
   cout << "Enter Your First String: ";
   getline(cin, str1);  // Read the entire line into
'str1'

   cout << "Enter Your Second String: ";
   getline(cin, str2);  // Read the entire line into
'str2'

   cout << "\nStr1 value Before Concatenate: " <<
str1;
   cout << "\nStr2 value Before Concatenate: " <<
str2;

   // Concatenate str2 to str1
   str1 += str2;

   cout << "\n\nStr1 value After Concatenate: " <<
str1;
   cout << "\nStr2 value After Concatenate: " <<
str2;
   return 0;
}
```

String Comparison

```
#include <iostream>
#include <string>  // For std::string

using namespace std;

int main()
{
    string str1, str2;
    cout << "Enter Your First String: ";
    getline(cin, str1);  // Read the entire line into 'str1'

    cout << "Enter Your Second String: ";
    getline(cin, str2);  // Read the entire line into 'str2'

    if (str1 == str2)
    {
        cout << "str1 and str2 are equal";
    }
    else
    {
        cout << "str1 and str2 are not equal";
    }
    return 0;
}
```

OOPS

C++ What is OOP?
OOP stands for Object-Oriented Programming.

C++ What are Classes and Objects?

Classes and objects are the two main aspects of object-oriented programming.

Look at the following illustration to see the difference between class and objects:

class

Fruit

objects

Apple

Banana

Mango

example:
class
Car
objects
Volvo
Audi
Toyota

```
#include<iostream>
using namespace std;
class person //class name//
{
	int age;
	char name[30];
	public:
		input_details()//create function
		{
			cout<<"enter your name"<<endl;
			cin>>name;
			cout<<"enter your age"<<endl;
			cin>>age;
		}
		 display_details()//create function
		{
			cout<<"name="<<name<<endl;
			cout<<"age=:"<<age<<endl;
		}
};

int main()
{
	person  p;//class name person last object create p//
	p.input_details();
	p.display_details();


	return 0;
}
```

Class Methods

Methods are **functions** that belongs to the class.
There are two ways to define functions that belongs to a class:

- Inside class definition
- Outside class definition

In the following example, we define a function inside the class, and we name it "myMethod".
Note: You access methods just like you access attributes; by creating an object of the class and using the dot syntax (.):

Inside class definition

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

class MyClass {         // The class
  public:               // Access specifier
    myMethod() {   // Method/function
     cout << "Hello World!";
   }
};

int main() {
  MyClass myObj;     // Create an object of MyClass
  myObj.myMethod();  // Call the method
  return 0;
```

Outside class definition

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

class MyClass {         // The class
  public:               // Access specifier
    myMethod();    // Method/function declaration
};

// Method/function definition outside the class
MyClass::myMethod() {
  cout << "Hello World!";
}

int main() {
  MyClass myObj;     // Create an object of MyClass
  myObj.myMethod();  // Call the method
  return 0;
}
```

//ARRAY OF OBJECT AND CLASS//

```
#include<iostream>
using namespace std;
class emp //class name//
{
	int id,salary;
	char name[50];
	public:

		input()//create function
		{
			cout<<"enter id or name or salary"<<endl;
			cin>>id>>name>>salary;

		}
		show()
		{
			cout<<"emp id"<<id<<endl;
			cout<<"emp name"<<name<<endl;
			cout<<"emp  salary"<<salary<<endl;
		}
};
int main()
{
	emp ob[3]; // ob bject create rray size empclass name //int i int loop run//ob][i].input() input use//ob][i].show(); outpue show//
	int i;
	for(i=0;i<3;i++){
		ob[i].input(); //input print//
	}
	for(i=0;i<3;i++){
		ob[i].show();//output print//
	}
	return 0;
}
```

Constructors

A constructor in C++ is a **special method** that is automatically called when an object of a class is created.
To create a constructor, use the same name as the class, followed by parentheses ():

//constructor

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

class MyClass {     // The class
  public:           // Access specifier
    MyClass() {     // Constructor
      cout << "Hello World!";
    }
};

int main() {
  MyClass myObj;// Create an object of MyClass (this will call the constructor)
  return 0;
}
```

What is a destructor in C++?

An equivalent special member function to a constructor is a destructor. The constructor creates class objects, which are destroyed by the destructor. The word "destructor," followed by the tilde () symbol, is the same as the class name. You can only define one destructor at a time. One method of destroying an object made by a constructor is to use a destructor. Destructors cannot be overloaded as a result. Destructors don't take any arguments and don't give anything back. As soon as the item leaves the scope, it is immediately called. Destructors free up the memory used by the objects the constructor generated. Destructor reverses the process of creating things by destroying them.
The language used to define the class's destructor

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

//DESCRUCTOR//

```
#include <iostream>
using namespace std;
class test{
    int a,b;
    public:
        test()
        {
            cout<<"test class object created:";
        }
    //define destructor using  ~sign.
    ~test(){
        cout<<endl<<"test class object destroyed";
    }
};
int main() {
// create object//
  test testobj;

        return 0;
  }
```

Access Specifiers

By now, you are quite familiar with the public keyword that appears in all of our class examples:
he public keyword is an **access specifier.** Access specifiers define how the members (attributes and methods) of a class can be accessed. In the example above, the members are public - which means that they can be accessed and modified from outside the code.
However, what if we want members to be private and hidden from the outside world?
In C++, there are three access specifiers:

- public - members are accessible from outside the class
- private - members cannot be accessed (or viewed) from outside the class
- protected - members cannot be accessed from outside the class, however, they can be accessed in inherited classes. You will learn more about Inheritance later.

In the following example, we demonstrate the differences between public and private members:

public Access Modifier

- The public keyword is used to create public members (data and functions).
- The public members are accessible from any part of the program.

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

// define a class
class Sample {

   // public elements
  public:
   int age;

   displayAge() {
          cout << "Enter your age: ";
          cin>>age;

   }

   dispiout(){

      cout<<age;

      }
};

int main() {

   // declare a class object
   Sample obj1;

   // call class function
   obj1.displayAge();
   obj1.dispiout();


   return 0;
}
```

Private Access Modifier

- The private keyword is used to create private members (data and functions).
- The private members can only be accessed from within the class.
- However, friend classes and friend functions can access private members.

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

// define a class
class Sample {
      // private elements
   private:
    int age;
    // public elements
   public:
    displayAge() {
          cout << "Enter your age: ";
          cin>>age;
    }
    dispiout(){
      cout<<age;

      }
};
int main() {
    // declare a class object
    Sample obj1;
    // call class function
    obj1.displayAge();
    obj1.dispiout();

    return 0;
}

```

Protected Access Modifier

Before we learn about the protected access specifier, make sure you know about inheritance in C++.

- The protected keyword is used to create protected members (data and function).
- The protected members can be accessed within the class and from the derived class.

```
#include <iostream>
using namespace std;
// define a class
class Sample {
   // protected elements
   protected:
    int age;
    // public elements
   public:
    displayAge() {
          cout << "Enter your age: ";
          cin>>age;
    }
    dispiout(){
      cout<<age;
      }
};
int main() {

    // declare a class object
    Sample obj1;

    // call class function
    obj1.displayAge();
    obj1.dispiout();


    return 0;
}
```

Summary: public, private, and protected

- public elements can be accessed by all other classes and functions.
- private elements cannot be accessed outside the class in which they are declared, except by friend classes and functions.
- protected elements are just like the private, except they can be accessed by derived classes.

Specifiers	Same Class	Derived Class	Outside Class
public	Yes	Yes	Yes
private	Yes	No	No
protected	Yes	Yes	No

C++ Encapsulation

Encapsulation is one of the key features of object-oriented programming. It involves the bundling of data members and functions inside a single class.

Bundling similar data members and functions inside a class together also helps in data hiding.

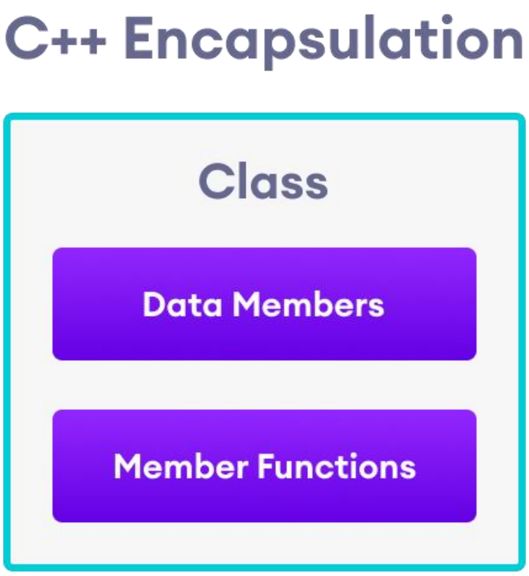
C++ Encapsulation

In general, encapsulation is a process of wrapping similar code in one place.

In C++, we can bundle data members and functions that operate together inside a single class. For example,

```
class Rectangle {
  public:
    int length;
    int breadth;

    int getArea() {
      return length * breadth;
    }
};
```

In the above program, the function getArea() calculates the area of a rectangle. To calculate the area, it needs length and breadth. Hence, the data members (length and breadth) and the function getArea() are kept together in the Rectangle class.



```
// Program to calculate the area of a rectangle
#include <iostream>
using namespace std;

class Rectangle {
  public:
    // Variables required for area calculation
    int length;
    int breadth;

    // Constructor to initialize variables
    Rectangle(int len, int brth) : length(len), breadth(brth)
{}

    // Function to calculate area
    int getArea() {
      return length * breadth;
    }
};

int main() {
  // Create object of Rectangle class
  Rectangle rect(8, 6);

  // Call getArea() function
  cout << "Area = " << rect.getArea();

  return 0;
}
```

C++ friend Function and friend Classes

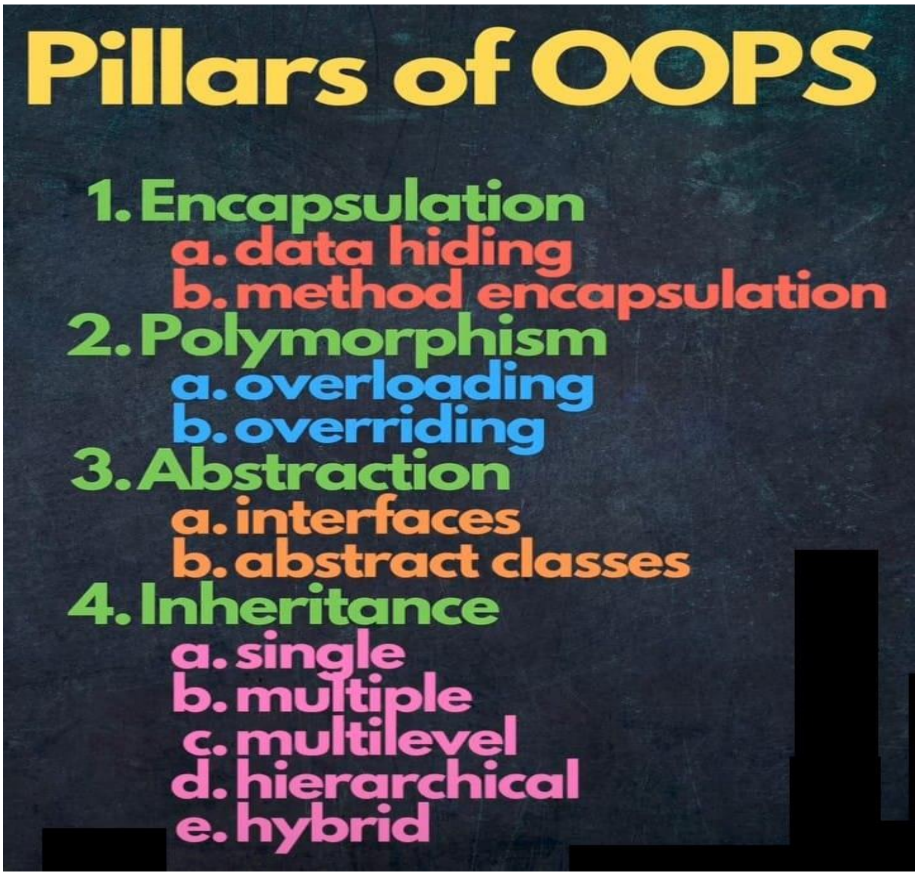
Data hiding is a fundamental concept of [object-oriented programming](). It restricts the access of private members from outside of the class.

Similarly, protected members can only be accessed by derived classes and are inaccessible from outside. For example,

A **friend function** can access the **private** and **protected** data of a class. We declare a friend function using the friend keyword inside the body of the class.

```
class className {
    ... .. ...
    friend returnType functionName(arguments);
    ... .. ...
}
```

```
//friend function//
#include <iostream>
using namespace std;
class A{
	int a,b;
	public:
		void input()
		{
			cout<<"enter  value a and b:";
			cin>>a>>b;
		}
		friend void add(A ob); //a class ob bject create //
};
void add(A ob){
 int c;
 c=ob.a+ob.b;
 cout<<"sum="<<c;
}
int main() {
  A kk;
  kk.input();
  add(kk);
		return 0;
   }
```



C++ Inheritance

Inheritance is one of the key features of [Object-oriented programming in C++]. It allows us to create a new [class] (derived class) from an existing class (base class).

The derived class inherits the features from the base class and can have additional features of its own. For example,

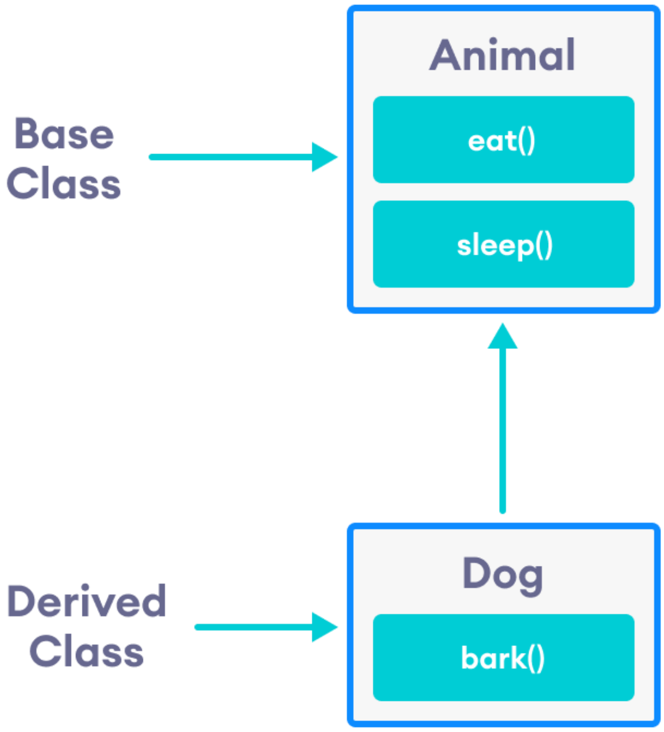
C++ Inheritance

Inheritance is one of the key features of [Object-oriented programming in C++]. It allows us to create a new [class] (derived class) from an existing class (base class).

The derived class inherits the features from the base class and can have additional features of its own. For example,

```
class Animal {
    // eat() function
    // sleep() function
};

class Dog : public Animal {
    // bark() function
};
```

Here, the Dog class is derived from the Animal class. Since Dog is derived from Animal, members of Animal are accessible to Dog.



Inheritance in C++
Notice the use of the keyword public while inheriting Dog from Animal.
class Dog : public Animal {...};
We can also use the keywords private and protected instead of public. We will learn about the differences between using private, public and protected later in this tutorial.

is-a relationship
Inheritance is an **is-a relationship**. We use inheritance only if an **is-a relationship** is present between the two classes.
Here are some examples:

- A car is a vehicle.
- Orange is a fruit.
- A surgeon is a doctor.
- A dog is an animal.

Example 1: Simple Example of C++ Inheritance

```
#include <iostream>
using namespace std;
// base class
class Animal {
   public:
     eat() {
        cout << "I can eat!" << endl;
    }
     sleep() {
        cout << "I can sleep!" << endl;
    }
};
// derived class
class Dog : public Animal {

   public:
     bark() {
        cout << "I can bark! Woof woof!!" << endl;
    }
};

int main() {
    // Create object of the Dog class
    Dog dog1;

    // Calling members of the base class
    dog1.eat();
    dog1.sleep();
// Calling member of the derived class
    dog1.bark();

    return 0;
}
```

C++ protected Members
The access modifier protected is especially relevant when it comes to C++ inheritance.
Like private members, protected members are inaccessible outside of the class. However, they can be accessed by **derived classes** and **friend classes/functions**.
We need protected members if we want to hide the data of a class, but still want that data to be inherited by its derived classes.
To learn more about protected, refer to our C++ Access Modifiers tutorial.

Example 2 : C++ protected Members
// C++ program to demonstrate protected members

```
#include <iostream>
#include <string>
using namespace std;
// base class
class data {
   private:
   int a;
   protected:
    string name;
   public:
     no() {
        cout << "enter no " << endl;
        cin>>a;
    }
    nameen() {
        cout << "enter name" << endl;
        cin>>name;
    }
    output(){
      cout<<a<< endl;
      cout<<name<<endl;
      }
};
// derived class
class Doo : public data {
      private:
      int b;
       public:
      no2() {
        cout << "enter no2 " << endl;
        cin>>b;
    }
    ouput3(){
      cout<<b;
      }
```

```
};
int main() {
    // Create object of the Dog class
    Doo doo1;

    // Calling members of the base class
    doo1.no();
    doo1.nameen();
    doo1.output();

    // Calling member of the derived class
    doo1.no2();
    doo1.ouput3();
    return 0;
}
```

Access Modes in C++ Inheritance

In our previous tutorials, we have learned about C++ access specifiers such as public, private, and protected.

So far, we have used the public keyword in order to inherit a class from a previously-existing base class. However, we can also use the private and protected keywords to inherit classes. For example,

```
class Animal {
    // code
};

class Dog : private Animal {
    // code
};
class Cat : protected Animal {
    // code
};
```

The various ways we can derive classes are known as **access modes**. These access modes have the following effect:

1. **public:** If a derived class is declared in public mode, then the members of the base class are inherited by the derived class just as they are.
2. **private:** In this case, all the members of the base class become private members in the derived class.
3. **protected:** The public members of the base class become protected members in the derived class.

The private members of the base class are always private in the derived class.

To learn more, visit our [C++ public, private, protected inheritance] tutorial.

Inheritance

In C++, it is possible to inherit attributes and methods from one class to another. We group the "inheritance concept" into two categories:

- **derived class** (child) - the class that inherits from another class
- **base class** (parent) - the class being inherited from

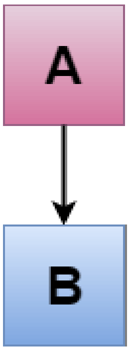
To inherit from a class, use the : symbol.

In the example below, the Car class (child) inherits the attributes and methods from the Vehicle class (parent):

Example

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.

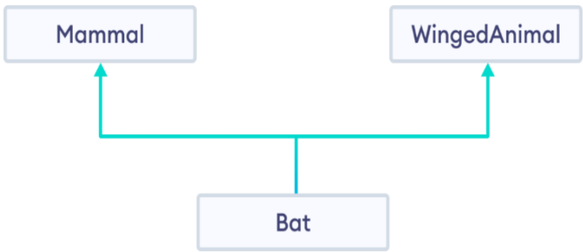
```
// single in herit//
#include <iostream>
using namespace std;
 class Base
 {
        protected:
        int a,b;
        public:
            show()
            {
                cout<<"enter a"<<endl;
                cin>>a;
                cout<<"enter b"<<endl;
```

```
            cin>>b;
        }
         out()
        {
            cout<<a<<ends<<b<<endl;
        }
};
class number:public Base// base is class name // derive
class and base  class ko  bi inherit  karega //
{
    private:
        int m,n;
    public:
        disp()
        {
            cout<<"enter m and n";
            cin>>m>>n;
        }
      jj()
        {
            cout<<m<<ends<<n<<endl;
        }

};

int main()
{

    //derive class to base class call in
object//
    number obj;  //object name =obj//
    obj.show();//disp member fuction
call//
    obj.out();

    obj.disp();
    obj.jj();

   return 0;
```

```
}


```

C++ Multiple Inheritance

In C++ programming, a class can be derived from more than one parent. For example, A class Bat is derived from base classes Mammal and WingedAnimal. It makes sense because bat is a mammal as well as a winged animal.



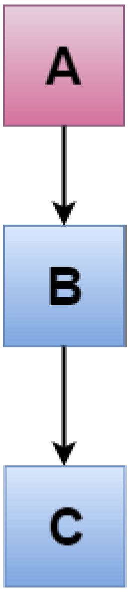
Multiple Inheritance

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

class Mammal {
public:
    Mammal() {
        cout << "Mammals can give direct birth." << endl;
    }
};

class WingedAnimal {
public:
```

```
    WingedAnimal() {
        cout << "Winged animal can flap." << endl;
    }
};

class Bat: public Mammal, public WingedAnimal {};

int main() {
    Bat b1;
    return 0;
}
```

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.

//multilevel in herit
#include <iostream>

```
using namespace std;

 class base{
                              private:
                                  int a;
                                  public:
                                        input()
                                       {
                                             cout<<"enter no a=";
                                             cin>>a;
                                        }
   show()
  {
                              cout<<"a="<<a<<endl;

}
 };

 class derive1:public base//public/private/protected
base//any derive name//
 {
                              private:
                                  int b;
                                  public:
                                       void input1(){

                                             cout<<"enter no b=";
                                             cin>>b;

                                        }

                                        void show1()
                                        {
                                             cout<<"b="<<b<<endl;
                                        }
 };

class derive2:public derive1
 {
```

```
            private:
                int c;
                public:
                    input2(){
                            cout<<"enter no c=";
                        cin>>c;

                    }

                    show2()
                    {
                        cout<<"c="<<c<<endl;
                    }

};
int main()
{
            base ob;
            ob.input();
            ob.show();

            derive1 ob1;
            ob1.input1();
            ob1.show1();

            derive2 ob2;
            ob2.input2();
            ob2.show2();

            return 0;
}
```

C++ Hierarchical Inheritance

If more than one class is inherited from the base class, it's known as [hierarchical inheritance]. In hierarchical inheritance, all features that are common in child classes are included in the base class.

For example, Physics, Chemistry, Biology are derived from Science class. Similarly, Dog, Cat, Horse are derived from Animal class.

Syntax of Hierarchical Inheritance

```
class base_class {
    ... .. ...
}

class first_derived_class: public base_class {
    ... .. ...
}

class second_derived_class: public base_class {
    ... .. ...
}

class third_derived_class: public base_class {
    ... .. ...
}
```

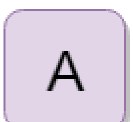
// C++ program to demonstrate hierarchical inheritance

```
#include <iostream>
using namespace std;
// base class
class Animal {
public:
    info() {
```

```
        cout << "I am an animal." << endl;
    }
};
// derived class 1
class Dog : public Animal {
public:
    bark() {
        cout << "I am a Dog. Woof woof." << endl;
    }
};
// derived class 2
class Cat : public Animal {
public:
     meow() {
        cout << "I am a Cat. Meow." << endl;
    }
};
int main() {
    // create object of Dog class
    Dog dog1;
    cout << "Dog Class:" << endl;
    dog1.info();  // parent Class function
    dog1.bark();
    // create object of Cat class
    Cat cat1;
    cout << "\nCat Class:" << endl;
    cat1.info();  // parent Class function
    cat1.meow();
    return 0;
}
```

C++ Hybrid Inheritance

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



```cpp
#include <iostream>
using namespace std;
// Base class
class Base {
public:
    void displayBase() {
        cout << "This is the Base class." << endl;
    }
};
// Derived class from Base (Single Inheritance)
class Derived1 : public Base {
```

```
public:
    void displayDerived1() {
        cout << "This is the Derived1 class." << endl;
    }
};
// Another base class
class AnotherBase {
public:
    void displayAnotherBase() {
        cout << "This is the AnotherBase class." << endl;
    }
};
// Derived class from both Base and AnotherBase (Multiple Inheritance)
class Derived2 : public Derived1, public AnotherBase {
public:
    void displayDerived2() {
        cout << "This is the Derived2 class." << endl;
    }
};
int main() {
    Derived2 obj;
    // Accessing methods from Base class through Derived2
    obj.displayBase();        // From Base class
    obj.displayDerived1();    // From Derived1 class
    obj.displayAnotherBase(); // From AnotherBase class
    obj.displayDerived2();    // From Derived2 class
    return 0;
}
```

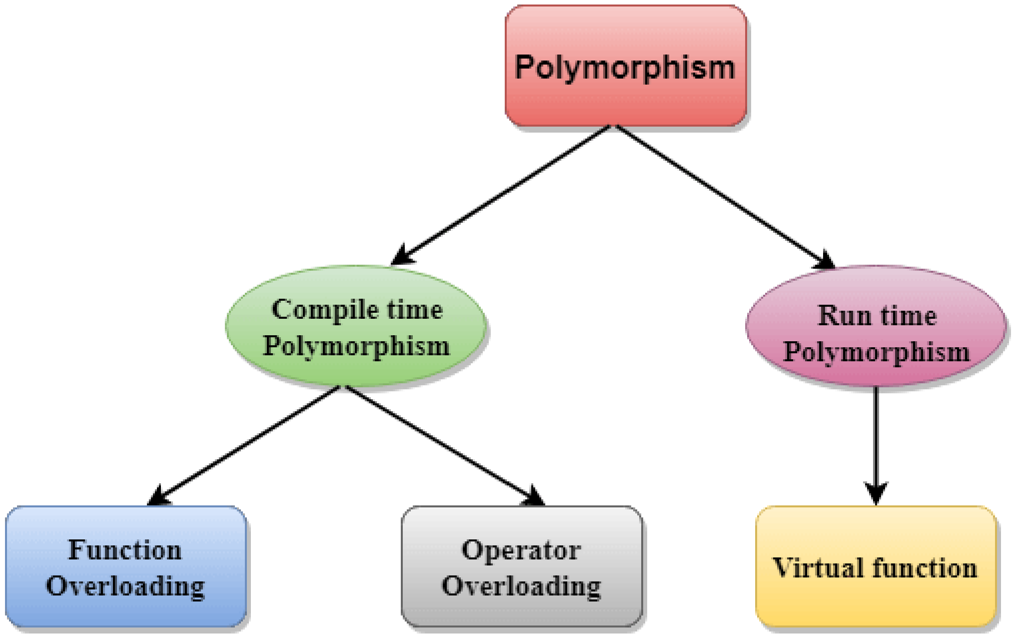
C++ Polymorphism

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.

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

```
        cout<<"Class B";
  }
};
```

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

- **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.

C++ Runtime Polymorphism Example

Let's see a simple example of run time polymorphism in C++.
// an example without the virtual keyword.

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

C++ Overloading (Function and Operator)

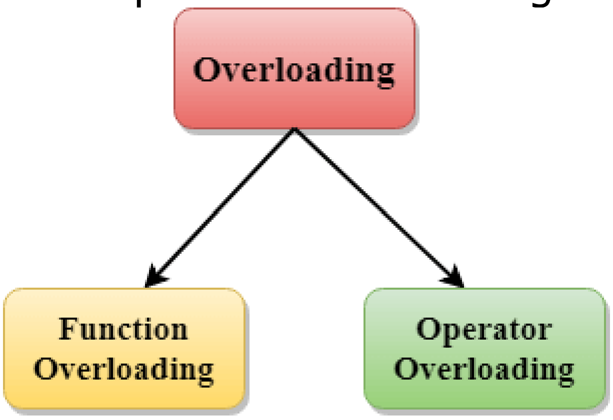
If we create two or more members having the same name but different in number or type of parameter, it is known as C++ overloading. In C++, we can overload:

- 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++. In function overloading, the function is redefined by using either different types of arguments or a different number of arguments. It is only through these differences compiler can differentiate between the functions.

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.

```cpp
#include <iostream>
using namespace std;
class Cal {
    public:
static int add(int a,int b){
        return a + b;
    }
static int add(int a, int b, int c)
    {
        return a + b + c;
    }
};
int main(void) {
    Cal C;                          //     class object declaration.
    cout<<C.add(10, 20)<<endl;
    cout<<C.add(12, 20, 23);
   return 0;
}
```

// Program of function overloading with different types of arguments

```
#include<iostream>
using namespace std;
int mul(int,int);
float mul(float,int);


int mul(int a,int b)
{
    return a*b;
}
float mul(double x, int y)
{
    return x*y;
}
int main()
{
    int r1 = mul(6,7);
    float r2 = mul(0.2,3);
    std::cout << "r1 is : " <<r1<< std::endl;
    std::cout <<"r2 is : "  <<r2<< std::endl;
    return 0;
}
```

// C++ program to demonstrate function overriding

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

class Base {
   public:
    void print() {
        cout << "Base Function" << endl;
    }
};

class Derived : public Base {
   public:
    void print() {
        cout << "Derived Function" << endl;
    }
};

int main() {
    Derived derived1;
    derived1.print();
    return 0;
}
```

```cpp
// C++ program to demonstrate function overriding

#include <iostream>
using namespace std;

class Base {
   public:
    void print() {
        cout << "Base Function" << endl;
    }
};

class Derived : public Base {
   public:
    void print() {
        cout << "Derived Function" << endl;
    }
};

int main() {
    Derived derived1;
    derived1.print();
    return 0;
}
```

```
// C++ program to call the overridden function
// from a member function of the derived class

#include <iostream>
using namespace std;

class Base {
   public:
    void print() {
        cout << "Base Function" << endl;
    }
};

class Derived : public Base {
   public:
    void print() {
        cout << "Derived Function" << endl;

        // call overridden function
        Base::print();
    }
};

int main() {
    Derived derived1;
    derived1.print();
    return 0;
}
```

```
// C++ program to access overridden function using pointer
// of Base type that points to an object of Derived class

#include <iostream>
using namespace std;

class Base {
   public:
    void print() {
        cout << "Base Function" << endl;
    }
};

class Derived : public Base {
   public:
    void print() {
        cout << "Derived Function" << endl;
    }
};

int main() {
    Derived derived1;

    // pointer of Base type that points to derived1
    Base* ptr = &derived1;

    // call function of Base class using ptr
    ptr->print();

    return 0;
}
```

C++ Operator Overloading

In C++, we can overload an operator as long as we are operating on user-defined types like objects or structures.

We cannot use operator overloading for basic types such as int, double, etc.

Operator overloading is basically function overloading, where different operator functions have the same symbol but different operands.

And, depending on the operands, different operator functions are executed. For example,

----

// C++ program to overload ++ when used as prefix

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

class Count {
   private:
    int value;

   public:

    // Constructor to initialize count to 5
    Count() : value(5) {}

    // Overload ++ when used as prefix
    void operator ++() {
        value = value + 1;
    }

    void display() {
        cout << "Count: " << value << endl;
    }
};

int main() {
    Count count1;

    // Call the "void operator ++()" function
    ++count1;

    count1.display();
    return 0;
}
```

C++ Function Overloading

In C++, we can use two functions having the same name if they have different parameters (either types or number of arguments).

And, depending upon the number/type of arguments, different functions are called. For example,

// C++ program to overload sum() function

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

// Function with 2 int parameters
int sum(int num1, int num2) {
    return num1 + num2;
}

// Function with 2 double parameters
double sum(double num1, double num2) {
    return num1 + num2;
}

// Function with 3 int parameters
int sum(int num1, int num2, int num3) {
    return num1 + num2 + num3;
}

int main() {
    // Call function with 2 int parameters
    cout << "Sum 1 = " << sum(5, 6) << endl;

    // Call function with 2 double parameters
    cout << "Sum 2 = " << sum(5.5, 6.6) << endl;

    // Call function with 3 int parameters
    cout << "Sum 3 = " << sum(5, 6, 7) << endl;

    return 0;
}
```

C++ Virtual Functions

In C++, we may not be able to override functions if we use a pointer of the base class to point to an object of the derived class. Using virtual functions in the base class ensures that the function can be overridden in these cases.

Thus, virtual functions actually fall under function overriding. For example,

// C++ program to demonstrate the use of virtual functions

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

class Base {
  public:
   virtual void print() {
       cout << "Base Function" << endl;
   }
};

class Derived : public Base {
  public:
   void print() {
       cout << "Derived Function" << endl;
   }
};

int main() {
   Derived derived1;

   // pointer of Base type that points to derived1
   Base* base1 = &derived1;

   // calls member function of Derived class
   base1->print();

   return 0;
}
```

Encapsulation

The meaning of **Encapsulation**, is to make sure that "sensitive" data is hidden from users. To achieve this, you must declare class variables/attributes as private (cannot be accessed from outside the class). If you want others to read or modify the value of a private member, you can provide public **get** and **set** methods.

Access Private Members

To access a private attribute, use public "get" and "set" methods:

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

class Employee {
  private:
    int salary;

  public:
    void setSalary(int s) {
      salary = s;
    }
    int getSalary() {
      return salary;
    }
};

int main() {
  Employee myObj;
  myObj.setSalary(50000);
  cout << myObj.getSalary();
  return 0;
}
```

example explained
The salary attribute is private, which have restricted access.
The public setSalary() method takes a parameter (s) and assigns it to the salary attribute (salary = s).
The public getSalary() method returns the value of the private salary attribute.
Inside main(), we create an object of the Employee class. Now we can use the setSalary() method to set the value of the private attribute to 50000. Then we call the getSalary() method on the object to return the value.

Why Encapsulation?

- **It is considered good practice to declare your class attributes as private (as often as you can). Encapsulation ensures better control of your data, because you (or others) can change one part of the code without affecting other parts**
- **Increased security of data**

POINTER

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

int main()
{
    // declare variables
    int var1 = 3;
    int var2 = 24;
    int var3 = 17;

    // print address of var1
    cout << "Address of var1: "<< &var1 << endl;

    // print address of var2
    cout << "Address of var2: " << &var2 << endl;

    // print address of var3
    cout << "Address of var3: " << &var3 << endl;
}
```

```
#include <iostream>
using namespace std;
int main() {
   int var = 5;

   // declare pointer variable
   int* pointVar;

   // store address of var
   pointVar = &var;

   // print value of var
   cout << "var = " << var << endl;

   // print address of var
   cout << "Address of var (&var) = " << &var << endl
       << endl;

   // print pointer pointVar
   cout << "pointVar = " << pointVar << endl;

   // print the content of the address pointVar points to
   cout << "Content of the address pointed to by pointVar
(*pointVar) = " << *pointVar << endl;

   return 0;
}
```

OUSIDE PROGRAM

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

// Creating a class
class myclass {       // This is the class
  public:             // This is an Access specifier
   void mymethod();    // This is the method
   void jj();        //call in outside data type(void) function
name jj() ouside call two dot  :://
      void uu();
};


// defining the method outside the class
void myclass::mymethod(){          // first class name-
myclass::function name()//
  cout<<"Hi how are you today?";
}



// creating an object of the class
int main(){
    myclass myobject; // we create an object myobject
    myobject.mymethod(); // calling the method



     myobject.jj();
     myobject.uu();
    return 0;
}
```

```
//friend function//

#include <iostream>
using namespace std;

class A{
	int a,b;

	public:
		void input()
		{
			cout<<"enter  value a and b:";
			cin>>a>>b;
		}

		friend void add(A ob); //a class ob bject create //
};
void add(A ob){
 int c;
 c=ob.a+ob.b;
 cout<<"sum="<<c;
}

int main() {
  A kk;
  kk.input();
  add(kk);



           return 0;
    }
```