

C++ Programs

- 1.C++Programs
- 2.Fibonacci Series
- 3.Prime Number
- 4.Palindrome Number
- 5.Factorial
- 6.Armstrong Number
- 7.Sum of digits
- 8.Reverse Number
- 9.Swap Number
- 10.Matrix Multiplication
- 11.Decimal to Binary
- 12.Number in Characters
- 13.Alphabet Triangle
- 14.Number Triangle
- 15.Fibonacci Triangle
- 16.Char array to string in C++
- 17.Calculator Program in C++
- 18.Program to convert infix to postfix expression in C++ using the Stack Data Structure
- 19.C++ program to merge two unsorted arrays
- 20.C++ coin change program
- 21.C++ program to add two complex numbers using class
- 22.C++ program to find the GCD of two numbers
- 23.C++ program to find greatest of four numbers
- 24.Delete Operator in C++How to concatenate two strings in c++
- 25.Upcasting and Down-casting in C++
- 26.C++ Dijkstra Algorithm using the priority queue
- 27.Constructor overloading in C++
- 28.Default arguments in C++
- 29.Dynamic binding in C++
- 30.Dynamic memory allocation in C++
- 31.Fast input and output in C++
- 32.Hierarchical inheritance in C++

33.Hybrid inheritance in C++

34.Multiple Inheritance in C++

35.C++ Bitwise XOR Operator

36.Different Ways to Compare Strings in C++

37.Reverse an Array in C++

38.C++ date and time

39.Copy elision in C++

40.Array of sets in C++

41.Smart pointers in C++

42.Types of polymorphism in C++

43.Implementing the sets without C++ STL containers

44.Scope Resolution Operator in C++

45.Static Member Function in C++

46.Const keyword in C++

47.Memset in C++

48.Type Casting in C++

49.Binary Operator Overloading in C++

50.Binary Search in C++

C++ Programs

C++ programs are frequently asked in the interview. These programs can be asked from basics, array, string, pointer, linked list, file handling etc. Let's see the list of top c++ programs.

1) [Fibonacci Series](#)

Write a c++ program to print fibonacci series without using recursion and using recursion.

Input: 10

Output: 0 1 1 2 3 5 8 13 21 34

2) [Prime number](#)

Write a c++ program to check prime number.

Input: 17

Output: not prime number

Input: 57

Output: prime number

3) [Palindrome number](#)

Write a c++ program to check palindrome number.

Input: 121

Output: not palindrome number

Input: 113

Output: palindrome number

4) [Factorial](#)

Write a c++ program to print factorial of a number.

Input: 5

Output: 120

Input: 6

Output: 720

5) [Armstrong number](#)

Write a c++ program to check armstrong number.

Input: 371

Output: armstrong

Input: 342

Output: not armstrong

6) [Sum of Digits](#)

Write a c++ program to print sum of digits.

Input: 23

Output: 5

Input: 624

Output: 12

7) [Reverse Number](#)

Write a c++ program to reverse given number.

Input: 234

Output: 432

8) [Swap two numbers without using third variable](#)

Write a c++ program to swap two numbers without using third variable.

Input: a=5 b=10

Output: a=10 b=5

9) [Matrix Multiplication](#)

Write a c++ program to print multiplication of 2 matrices.

Input:

```
first matrix elements:
1 2 3
1 2 3
1 2 3
second matrix elements
1 1 1
2 1 2
3 2 1
```

Output:

```
multiplication of the matrix:
14 9 8
14 9 8
14 9 8
```

10) [Decimal to Binary](#)

Write a c++ program to convert decimal number to binary.

Input: 9

Output: 1001

Input: 20

Output: 10100

11) [Alphabet Triangle](#)

Write a c++ program to print alphabet triangle.

Output:

```
  A
 ABA
ABCBA
ABCD CBA
ABCDEDCBA
```

12) [Number Triangle](#)

Write a c++ program to print number triangle.

Input: 7

Output:

```
enter the range= 6
 1
 121
12321
1234321
123454321
12345654321
```

13) [Fibonacci Triangle](#)

Write a c++ program to generate fibonacci triangle.

Input: 5

Output:

```
1
1 1
1 1 2
1 1 2 3
1 1 2 3 5
```

14) [Number in Characters](#)

Write a c++ program to convert number in characters.

Input: 74254

Output:Seven Four Two Five Four

Input: 203

Output: two zero three

Fibonacci Series in C++

Fibonacci Series in C++: In case of fibonacci series, next number is the sum of previous two numbers for example 0, 1, 1, 2, 3, 5, 8, 13, 21 etc. The first two numbers of fibonacci series are 0 and 1.

There are two ways to write the fibonacci series program:

- Fibonacci Series without recursion
- Fibonacci Series using recursion

Fibonacci Series in C++ without Recursion

Let's see the fibonacci series program in C++ without recursion.

1. `#include <iostream>`
2. `using namespace std;`
3. `int main() {`
4. `int n1=0,n2=1,n3,i,number;`
5. `cout<<"Enter the number of elements: ";`
6. `cin>>number;`
7. `cout<<n1<<" "<n2<<" "; //printing 0 and 1`

```
8.  for(i=2;i<number;++i) //loop starts from 2 because 0 and 1 are already printed
9.  {
10.  n3=n1+n2;
11.  cout<<n3<<" ";
12.  n1=n2;
13.  n2=n3;
14. }
15. return 0;
16. }
```

Output:

```
Enter the number of elements: 10
0 1 1 2 3 5 8 13 21 34
```

Fibonnaci series using recursion in C++

Let's see the fibonacci series program in C++ using recursion.

```
1.  #include<iostream>
2.  using namespace std;
3.  void printFibonacci(int n){
4.      static int n1=0, n2=1, n3;
5.      if(n>0){
6.          n3 = n1 + n2;
7.          n1 = n2;
8.          n2 = n3;
9.      cout<<n3<<" ";
10.      printFibonacci(n-1);
11.  }
12. }
13. int main(){
14.     int n;
15.     cout<<"Enter the number of elements: ";
16.     cin>>n;
17.     cout<<"Fibonacci Series: ";
18.     cout<<"0 "<<"1 ";
19.     printFibonacci(n-2); //n-2 because 2 numbers are already printed
20.     return 0;
21. }
```

Output:

```
Enter the number of elements: 15
Fibonacci Series: 0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Prime Number Program in C++

Prime number is a number that is greater than 1 and divided by 1 or itself. In other words, prime numbers can't be divided by other numbers than itself or 1. For example 2, 3, 5, 7, 11, 13, 17, 19, 23.... are the prime numbers.

Let's see the prime number program in C++. In this C++ program, we will take an input from the user and check whether the number is prime or not.

```
1.  #include <iostream>
2.  using namespace std;
3.  int main()
4.  {
5.      int n, i, m=0, flag=0;
6.      cout << "Enter the Number to check Prime: ";
7.      cin >> n;
```

```

8.  m=n/2;
9.  for(i = 2; i <= m; i++)
10. {
11.    if(n % i == 0)
12.    {
13.      cout<<"Number is not Prime."<<endl;
14.      flag=1;
15.      break;
16.    }
17. }
18. if (flag==0)
19.   cout << "Number is Prime."<<endl;
20. return 0;
21.}

```

Output:

```

Enter the Number to check Prime: 17
Number is Prime.
Enter the Number to check Prime: 57
Number is not Prime.

```

Palindrome program in C++

A **palindrome number** is a number that is same after reverse. For example 121, 34543, 343, 131, 48984 are the palindrome numbers.

Palindrome number algorithm

- Get the number from user
- Hold the number in temporary variable
- Reverse the number
- Compare the temporary number with reversed number
- If both numbers are same, print palindrome number
- Else print not palindrome number

Let's see the palindrome program in C++. In this program, we will get an input from the user and check whether number is palindrome or not.

```

1. #include <iostream>
2. using namespace std;
3. int main()
4. {
5.   int n,r,sum=0,temp;
6.   cout<<"Enter the Number=";
7.   cin>>n;
8.   temp=n;
9.   while(n>0)
10. {
11.  r=n%10;
12.  sum=(sum*10)+r;
13.  n=n/10;
14. }
15. if(temp==sum)
16. cout<<"Number is Palindrome.";
17. else
18. cout<<"Number is not Palindrome.";
19. return 0;
20.}

```

Output:

```
Enter the Number=121
Number is Palindrome.
Enter the number=113
Number is not Palindrome.
```

Factorial program in C++

Factorial Program in C++: Factorial of n is the product of all positive descending integers. Factorial of n is denoted by n!. For example:

1. $4! = 4 \times 3 \times 2 \times 1 = 24$
2. $6! = 6 \times 5 \times 4 \times 3 \times 2 \times 1 = 720$

Here, 4! is pronounced as "4 factorial", it is also called "4 bang" or "4 shriek".

The factorial is normally used in Combinations and Permutations (mathematics).

There are many ways to write the factorial program in C++ language. Let's see the 2 ways to write the factorial program.

- Factorial Program using loop
- Factorial Program using recursion

Factorial Program using Loop

Let's see the factorial Program in C++ using loop.

1. `#include <iostream>`
2. `using namespace std;`
3. `int main()`
4. `{`
5. `int i,fact=1,number;`
6. `cout<<"Enter any Number: ";`
7. `cin>>number;`
8. `for(i=1;i<=number;i++){`
9. `fact=fact*i;`
10. `}`
11. `cout<<"Factorial of " <<number<<" is: " <<fact<<endl;`
12. `return 0;`
13. `}`

Output:

```
Enter any Number: 5
Factorial of 5 is: 120
```

Factorial Program using Recursion

Let's see the factorial program in C++ using recursion.

1. `#include<iostream>`
2. `using namespace std;`
3. `int main()`
4. `{`
5. `int factorial(int);`
6. `int fact,value;`
7. `cout<<"Enter any number: ";`
8. `cin>>value;`
9. `fact=factorial(value);`
10. `cout<<"Factorial of a number is: " <<fact<<endl;`


```
11. return 0;
12. }
13. int factorial(int n)
14. {
15. if(n<0)
16. return(-1); /*Wrong value*/
17. if(n==0)
18. return(1); /*Terminating condition*/
19. else
20. {
21. return(n*factorial(n-1));
22. }
23. }
```

Output:

```
Enter any number: 6
Factorial of a number is: 720
```

Armstrong Number in C++

Before going to write the C++ program to check whether the number is Armstrong or not, let's understand what is Armstrong number.

Armstrong number is a number that is equal to the sum of cubes of its digits. For example 0, 1, 153, 370, 371 and 407 are the Armstrong numbers.

Let's try to understand why **371** is an Armstrong number.

1. $371 = (3*3*3)+(7*7*7)+(1*1*1)$
2. where:
3. $(3*3*3)=27$
4. $(7*7*7)=343$
5. $(1*1*1)=1$
6. So:
7. $27+343+1=371$

Let's see the C++ program to check Armstrong Number.

```
1. #include <iostream>
2. using namespace std;
3. int main()
4. {
5. int n,r,sum=0,temp;
6. cout<<"Enter the Number= ";
7. cin>>n;
8. temp=n;
9. while(n>0)
10. {
11. r=n%10;
12. sum=sum+(r*r*r);
13. n=n/10;
14. }
15. if(temp==sum)
16. cout<<"Armstrong Number."<<endl;
17. else
18. cout<<"Not Armstrong Number."<<endl;
19. return 0;
20. }
```

Output:

```
Enter the Number= 371
Armstrong Number.
Enter the Number= 342
Not Armstrong Number.
```

Sum of digits program in C++

We can write the sum of digits program in C++ language by the help of loop and mathematical operation only.

Sum of digits algorithm

To get sum of each digit by C++ program, use the following algorithm:

- **Step 1:** Get number by user
- **Step 2:** Get the modulus/remainder of the number
- **Step 3:** sum the remainder of the number
- **Step 4:** Divide the number by 10
- **Step 5:** Repeat the step 2 while number is greater than 0.

Let's see the sum of digits program in C++.

```
1. #include <iostream>
2. using namespace std;
3. int main()
4. {
5.     int n,sum=0,m;
6.     cout<<"Enter a number: ";
7.     cin>>n;
8.     while(n>0)
9.     {
10. m=n%10;
11. sum=sum+m;
12. n=n/10;
13. }
14. cout<<"Sum is= "<<sum<<endl;
15. return 0;
16. }
```

Output:

```
Enter a number: 23
Sum is= 5
Enter a number: 624
Sum is= 12
```

C++ Program to reverse number

We can reverse a number in C++ using loop and arithmetic operators. In this program, we are getting number as input from the user and reversing that number.

Let's see a simple C++ example to reverse a given number.

```
1. #include <iostream>
2. using namespace std;
3. int main()
4. {
5.     int n, reverse=0, rem;
6.     cout<<"Enter a number: ";
7.     cin>>n;
8.     while(n!=0)
9.     {
10.     rem=n%10;
```

```
11.    reverse=reverse*10+rem;
12.    n/=10;
13. }
14. cout<<"Reversed Number: "<<reverse<<endl;
15. return 0;
16. }
```

Output:

```
Enter a number: 234
Reversed Number: 432
```

C++ Program to swap two numbers without third variable

We can swap two numbers without using third variable. There are two common ways to swap two numbers without using third variable:

1. By + and -
2. By * and /

Program 1: Using * and /

Let's see a simple C++ example to swap two numbers without using third variable.

```
1.  #include <iostream>
2.  using namespace std;
3.  int main()
4.  {
5.  int a=5, b=10;
6.  cout<<"Before swap a= "<<a<<" b= "<<b<<endl;
7.  a=a*b; //a=50 (5*10)
8.  b=a/b; //b=5 (50/10)
9.  a=a/b; //a=10 (50/5)
10. cout<<"After swap a= "<<a<<" b= "<<b<<endl;
11. return 0;
12. }
```

Output:

```
Before swap a= 5 b= 10
After swap a= 10 b= 5
```

Program 2: Using + and -

Let's see another example to swap two numbers using + and -.

```
1.  #include <iostream>
2.  using namespace std;
3.  int main()
4.  {
5.  int a=5, b=10;
6.  cout<<"Before swap a= "<<a<<" b= "<<b<<endl;
7.  a=a+b; //a=15 (5+10)
8.  b=a-b; //b=5 (15-10)
9.  a=a-b; //a=10 (15-5)
10. cout<<"After swap a= "<<a<<" b= "<<b<<endl;
11. return 0;
12. }
```

Output:

```
Before swap a= 5 b= 10
After swap a= 10 b= 5
```

Matrix multiplication in C++

We can add, subtract, multiply and divide 2 matrices. To do so, we are taking input from the user for row number, column number, first matrix elements and second matrix elements. Then we are performing multiplication on the matrices entered by the user.

In matrix multiplication first matrix one row element is multiplied by second matrix all column elements.

Let's try to understand the matrix multiplication of **3*3 and 3*3** matrices by the figure given below:

Matrix 1

$$\begin{Bmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{Bmatrix}$$

Matrix 2

$$\begin{Bmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{Bmatrix}$$

Matrix 1

*

Matrix 2

$$\begin{Bmatrix} 1*1+1*2+1*3 & 1*1+1*2+1*3 & 1*1+1*2+1*3 \\ 2*1+2*2+2*3 & 2*1+2*2+2*3 & 2*1+2*2+2*3 \\ 3*1+3*2+3*3 & 3*1+3*2+3*3 & 3*1+3*2+3*3 \end{Bmatrix}$$

Matrix 1

*

Matrix 2

$$\begin{Bmatrix} 6 & 6 & 6 \\ 12 & 12 & 12 \\ 18 & 18 & 18 \end{Bmatrix}$$

JavaTpoint

Let's see the program of matrix multiplication in C++.

```
1. #include <iostream>
2. using namespace std;
3. int main()
4. {
5.     int a[10][10],b[10][10],mul[10][10],r,c,i,j,k;
6.     cout<<"enter the number of row=";
7.     cin>>r;
8.     cout<<"enter the number of column=";
9.     cin>>c;
10. cout<<"enter the first matrix element=\n";
11. for(i=0;i<r;i++)
12. {
13.     for(j=0;j<c;j++)
14. {
15. cin>>a[i][j];
16. }
17. }
18. cout<<"enter the second matrix element=\n";
19. for(i=0;i<r;i++)
20. {
21.     for(j=0;j<c;j++)
22. {
23. cin>>b[i][j];
24. }
25. }
26. cout<<"multiply of the matrix=\n";
27. for(i=0;i<r;i++)
28. {
29.     for(j=0;j<c;j++)
30. {
31. mul[i][j]=0;
32.     for(k=0;k<c;k++)
```

```
33. {
34. mul[i][j] += a[i][k]*b[k][j];
35. }
36. }
37. }
38. //for printing result
39. for(i=0;i<r;i++)
40. {
41. for(j=0;j<c;j++)
42. {
43. cout<<mul[i][j]<<" ";
44. }
45. cout<<"\n";
46. }
47. return 0;
48. }
```

Output:

```
enter the number of row=3
enter the number of column=3
enter the first matrix element=
1 2 3
1 2 3
1 2 3
enter the second matrix element=
1 1 1
2 1 2
3 2 1
multiply of the matrix=
14 9 8
14 9 8
14 9 8
```

C++ Program to convert Decimal to Binary

We can convert any decimal number (base-10 (0 to 9)) into binary number (base-2 (0 or 1)) by C++ program.

Decimal Number

Decimal number is a base 10 number because it ranges from 0 to 9, there are total 10 digits between 0 to 9. Any combination of digits is decimal number such as 223, 585, 192, 0, 7 etc.

Binary Number

Binary number is a base 2 number because it is either 0 or 1. Any combination of 0 and 1 is binary number such as 1001, 101, 11111, 101010 etc.

Let's see the some binary numbers for the decimal number.

Decimal	Binary
1	0
2	10
3	11
4	100
5	101
6	110
7	111
8	1000

9	1001
10	1010

Decimal to Binary Conversion Algorithm

Step 1: Divide the number by 2 through % (modulus operator) and store the remainder in array

Step 2: Divide the number by 2 through / (division operator)

Step 3: Repeat the step 2 until the number is greater than zero

Let's see the C++ example to convert decimal to binary.

```
1. #include <iostream>
2. using namespace std;
3. int main()
4. {
5.     int a[10], n, i;
6.     cout<<"Enter the number to convert: ";
7.     cin>>n;
8.     for(i=0; n>0; i++)
9.     {
10.        a[i]=n%2;
11.        n= n/2;
12.    }
13.    cout<<"Binary of the given number= ";
14.    for(i=i-1 ;i>=0 ;i--)
15.    {
16.        cout<<a[i];
17.    }
18. }
```

Output:

```
Enter the number to convert: 9
Binary of the given number= 1001
```

C++ Program to Convert Number in Characters

In C++ language, we can easily convert number in characters by the help of loop and switch case. In this program, we are taking input from the user and iterating this number until it is 0. While iteration, we are dividing it by 10 and the remainder is passed in switch case to get the word for the number.

Let's see the C++ program to convert number in characters.

```
1. #include <iostream>
2. using namespace std;
3. int main()
4. {
5.     long int n,sum=0,r;
6.     cout<<"Enter the Number= ";
7.     cin>>n;
8.     while(n>0)
9.     {
10.        r=n%10;
11.        sum=sum*10+r;
12.        n=n/10;
13.    }
14.    n=sum;
```

```
15. while(n>0)
16. {
17. r=n%10;
18. switch(r)
19. {
20. case 1:
21. cout<<"one ";
22. break;
23. case 2:
24. cout<<"two ";
25. break;
26. case 3:
27. cout<<"three ";
28. break;
29. case 4:
30. cout<<"four ";
31. break;
32. case 5:
33. cout<<"five ";
34. break;
35. case 6:
36. cout<<"six ";
37. break;
38. case 7:
39. cout<<"seven ";
40. break;
41. case 8:
42. cout<<"eight ";
43. break;
44. case 9:
45. cout<<"nine ";
46. break;
47. case 0:
48. cout<<"zero ";
49. break;
50. default:
51. cout<<"ttt ";
52. break;
53. }
54. n=n/10;
55. }
56. }
```

Output:

```
Enter the Number= 74254
seven four two five four
```

C++ Program to Print Alphabet Triangle

There are different triangles that can be printed. Triangles can be generated by alphabets or numbers. In this C++ program, we are going to print alphabet triangles.

Let's see the C++ example to print alphabet triangle.

```
1. #include <iostream>
2. using namespace std;
3. int main()
4. {
```

```
5.  char ch='A';
6.  int i, j, k, m;
7.  for(i=1;i<=5;i++)
8.  {
9.      for(j=5;j>=i;j--)
10.         cout<<" ";
11.     for(k=1;k<=i;k++)
12.         cout<<ch++;
13.     ch--;
14.     for(m=1;m<i;m++)
15.         cout<<--ch;
16.     cout<<"\n";
17.     ch='A';
18. }
19. return 0;
20. }
```

Output:

```
A
ABA
ABCBA
ABCDcba
ABCDEDCBA
```

C++ Program to print Number Triangle

Like alphabet triangle, we can write the C++ program to print the number triangle. The number triangle can be printed in different ways.

Let's see the C++ example to print number triangle.

```
1. #include <iostream>
2. using namespace std;
3. int main()
4. {
5.     int i,j,k,l,n;
6.     cout<<"Enter the Range=";
7.     cin>>n;
8.     for(i=1;i<=n;i++)
9.     {
10.        for(j=1;j<=n-i;j++)
11.        {
12.            cout<<" ";
13.        }
14.        for(k=1;k<=i;k++)
15.        {
16.            cout<<k;
17.        }
18.        for(l=i-1;l>=1;l--)
19.        {
20.            cout<<l;
21.        }
22.        cout<<"\n";
23.    }
24.    return 0;
25. }
```

Output:

```
Enter the Range=5
```



```
1
121
12321
1234321
123454321
Enter the Range=6
1
121
2321
1234321
123454321
12345654321
```

C++ Program to generate Fibonacci Triangle

In this program, we are getting input from the user for the limit for fibonacci triangle, and printing the fibonacci series for the given number of times (limit).

Let's see the C++ example to generate fibonacci triangle.

1. `#include <iostream>`
2. `using namespace std;`
3. `int main()`
4. `{`
5. `int a=0,b=1,i,c,n,j;`
6. `cout<<"Enter the limit: ";`
7. `cin>>n;`
8. `for(i=1; i<=n; i++)`
9. `{`
10. `a=0;`
11. `b=1;`
12. `cout<<b<<"\t";`
13. `for(j=1; j<i; j++)`
14. `{`
15. `c=a+b;`
16. `cout<<c<<"\t";`
17. `a=b;`
18. `b=c;`
19. `}`
20. `cout<<"\n";`
21. `}`
22. `return 0;`
23. `}`

Output:

```
Enter the limit: 10
1
1      1
1      1      2
1      1      2      3
1      1      2      3      5
1      1      2      3      5      8
1      1      2      3      5      8      13
1      1      2      3      5      8      13      21
1      1      2      3      5      8      13      21      34
1      1      2      3      5      8      13      21
```

Char array to string in C++

"Char" data type or a character data type is used to store letters, unlike numbers and integers, which are stored in integer and floating-point or true-false value in Booleans.

Characters are integer type in nature, their size is 1-byte and printable characters are (space), !, " , #, \$, %, &, ' , (,), *, +, ,, -, ., /, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, :, ;, <, =, >, ?, @, A, a, B, b, C, c, D, d, E, e, F, f, G, g, H, h, I, i, J, j, K, k, L, l, M, m, N, n, O, o, P, p, Q, q, R, r, S, s, T, t, U, u, V, v, W, w, X, x, Y, y, Z, z, [, \], ^, _ , ` , { , | , } , ~ and DEL (delete).

We can initialize char variables using -

char ch2{ 'a' }; To print character "a".

char ch1{ 97 }; To print the value at code 97.

char ch{'5'}; To print number character "5".

C++ provides us with the following techniques to convert a char array to a string:

- Using 'c_str()' and 'strcpy()' function
- using a 'for' loop
- 'while' loop,
- '=' operator from the string class
- using custom function.

Syntax for using 'c_str()' and 'strcpy()' function is given below:

1. `"string-name.c_str();"`.

where, c_str() converts the contents of a string as C-style, non-terminated string. It gives direct access to the internal string buffer.

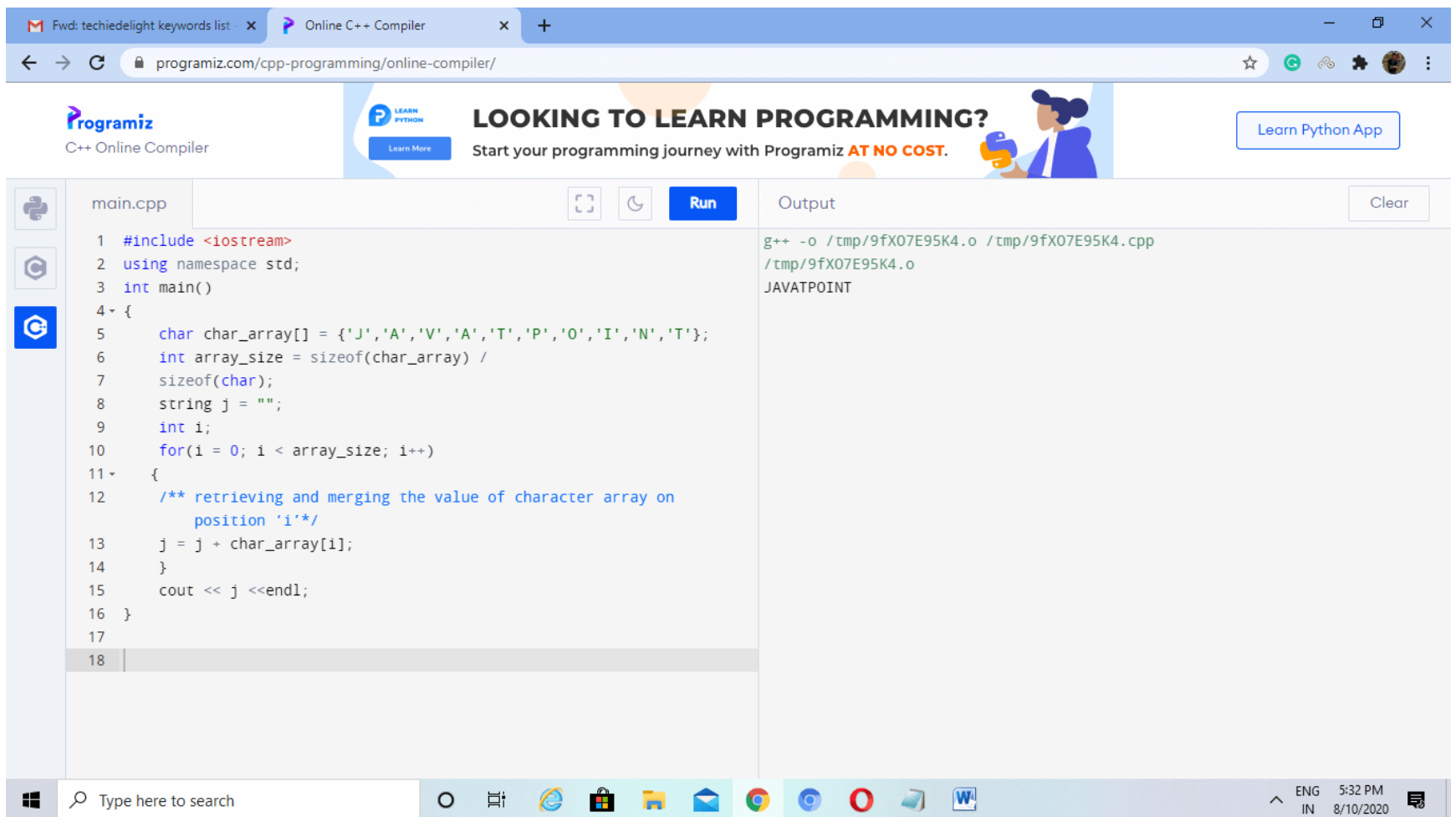
We can convert character to a string using 'for' a loop by -

First, declaring the Character Array and then assigning the size of the Array. Then, we declare two variables, one string type, and another int type. After this, we can use the 'for' loop by assigning 0 to the int variable where the int variable will have less value than array_size and we increase int variable value by one at every iteration. We have to store the value in the string variable on every iteration before displaying the string variable.

Code:-

```
1. #include <iostream>
2. using namespace std;
3. int main()
4. {
5.     char char_array[] = {'J','A','V','A','T','P','O','I','N','T'};
6.     int array_size = sizeof(char_array) /
7.     sizeof(char);
8.     string j = "";
9.     int i;
10.    for(i = 0; i < array_size; i++)
11.    {
12.        /** retrieving and merging the value of character array on position 'i'*/
13.        j = j + char_array[i];
14.    }
15.    cout << j << endl;
16. }
```

Output:



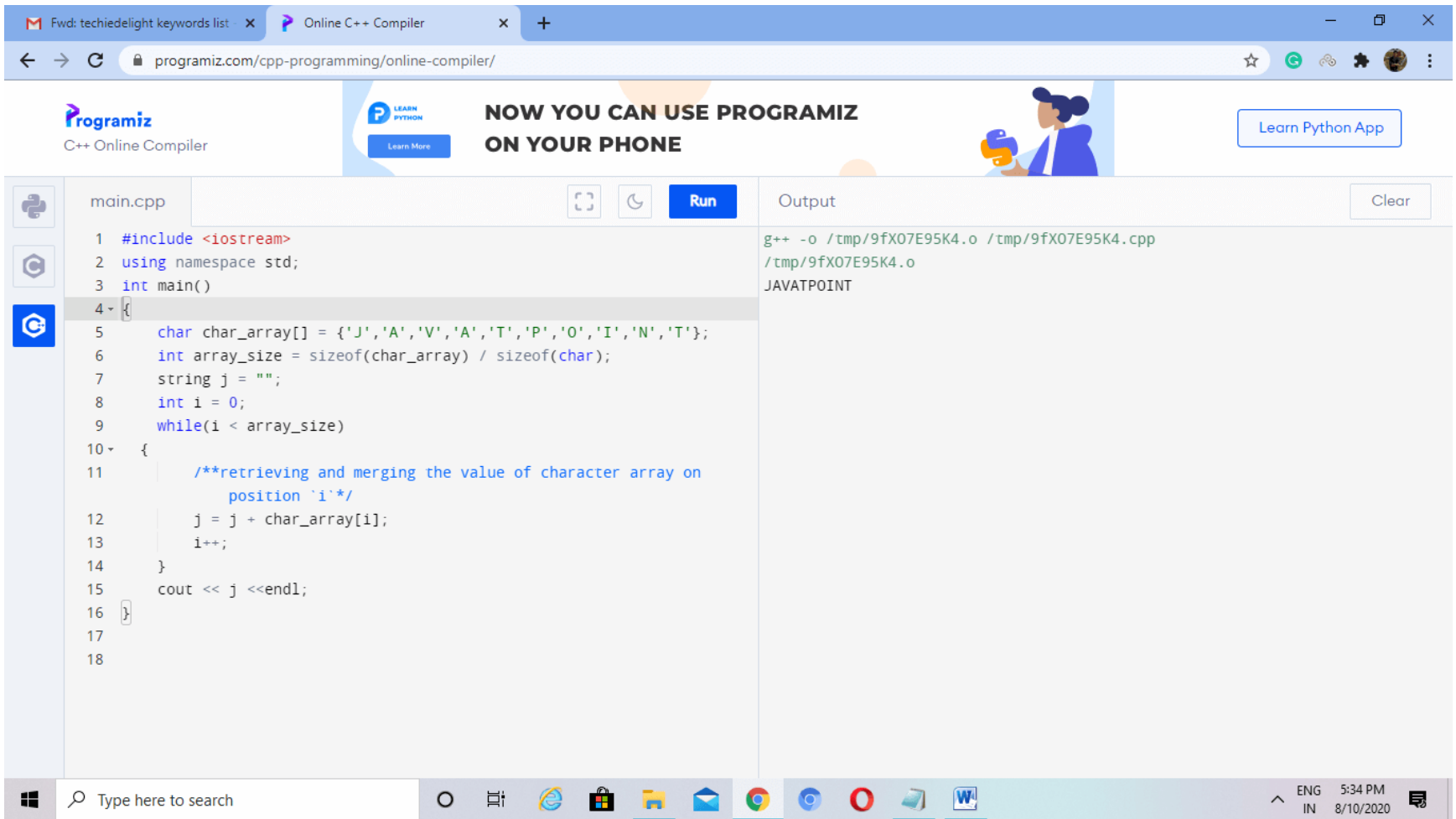
We can convert char to a string using 'while' loop by -

First declaring the Character Array and then assigning the size of the Array. Then, we declare two variables, one string type, and another int type with value 0. We use '[while](#) loop' to check int variable less than array_size on every iteration and store the value in a string variable before displaying the string variable.

Code:-

```
1. #include <iostream>
2. using namespace std;
3. int main()
4. {
5.     char char_array[] = {'J','A','V','A','T','P','O','I','N','T'};
6.     int array_size = sizeof(char_array) / sizeof(char);
7.     string j = "";
8.     int i = 0;
9.     while(i < array_size)
10. {
11.     /**retrieving and merging the value of character array on position `i`*/
12.     j = j + char_array[i];
13.     i++;
14. }
15. cout << j << endl;
16. }
```

Output:

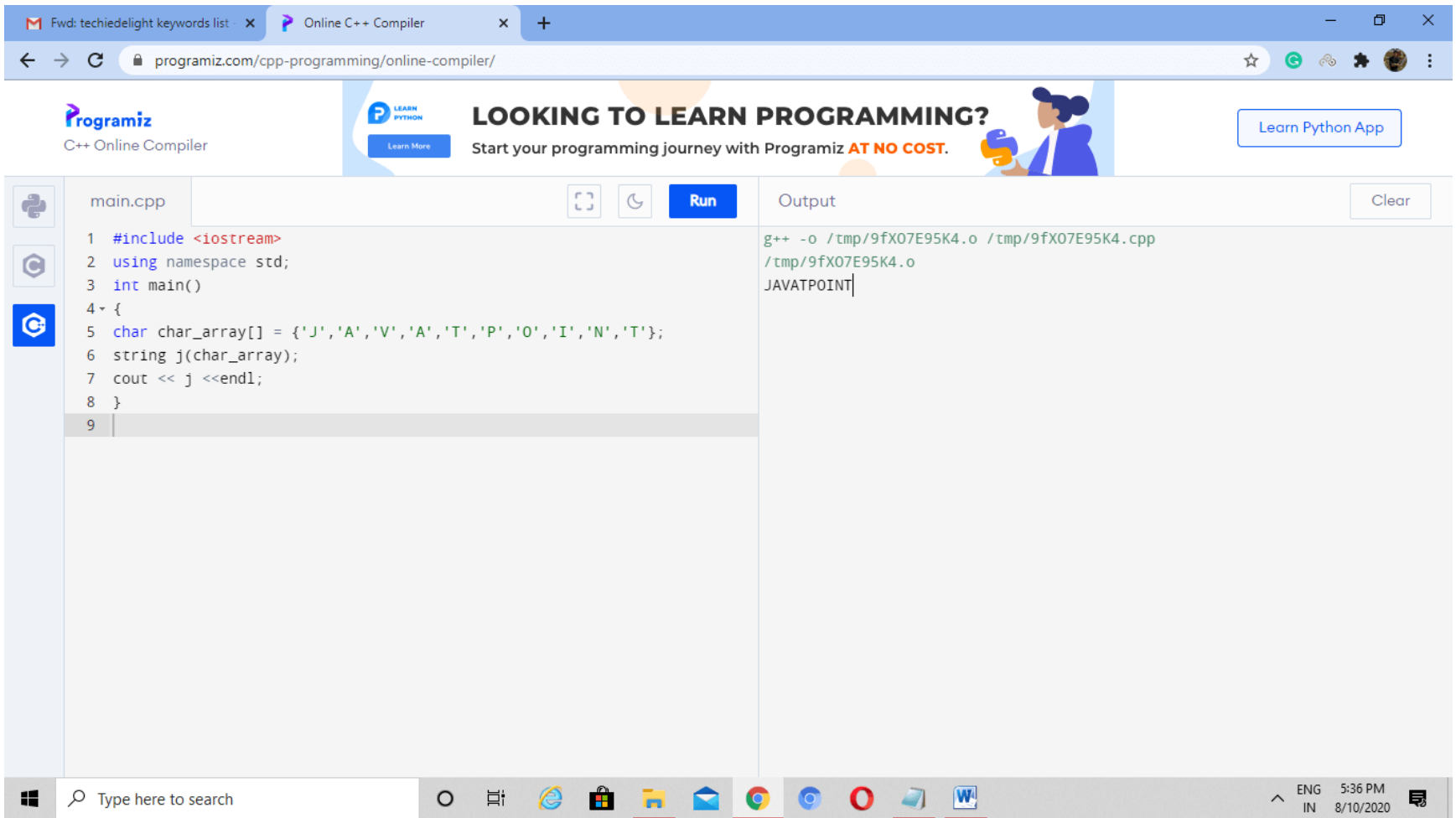


To convert character to a string using **std::string** constructor, we simply pass the array to string constructor.

Code:-

1. **#include <iostream>**
2. **using namespace std;**
3. **int main()**
4. **{**
5. **char** char_array[] = {'J','A','V','A','T','P','O','I','N','T'};
6. **string** j(char_array);
7. **cout << j <<endl;**
8. **}**

Output:

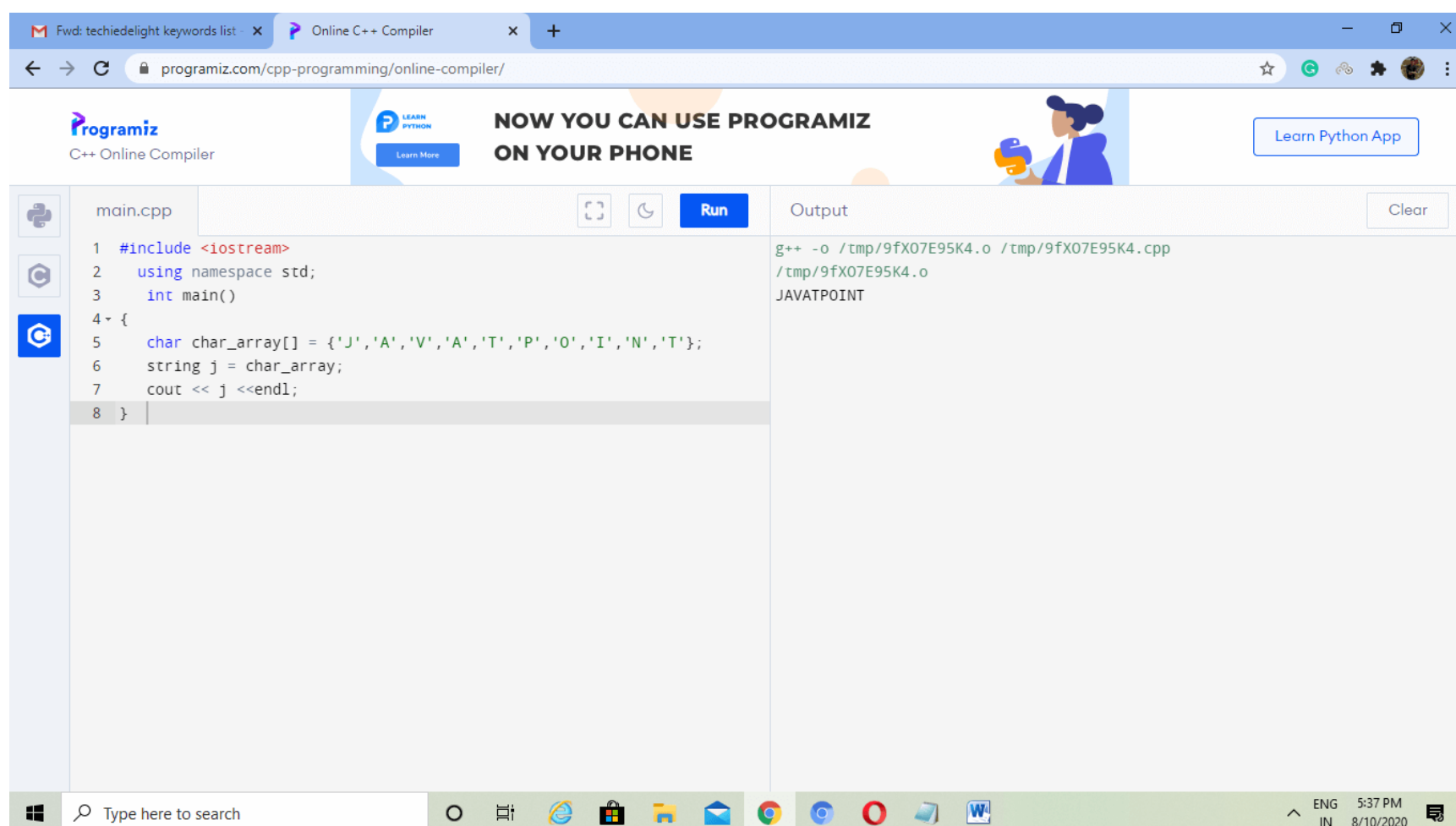


To convert a character array into a string using the '=' operator and string class, we have to pass character array to string variable.

Code:-

```
1. #include <iostream>
2. using namespace std;
3. int main()
4. {
5.     char char_array[] = {'J','A','V','A','T','P','O','I','N','T'};
6.     string j = char_array;
7.     cout << j << endl;
8. }
```

Output:



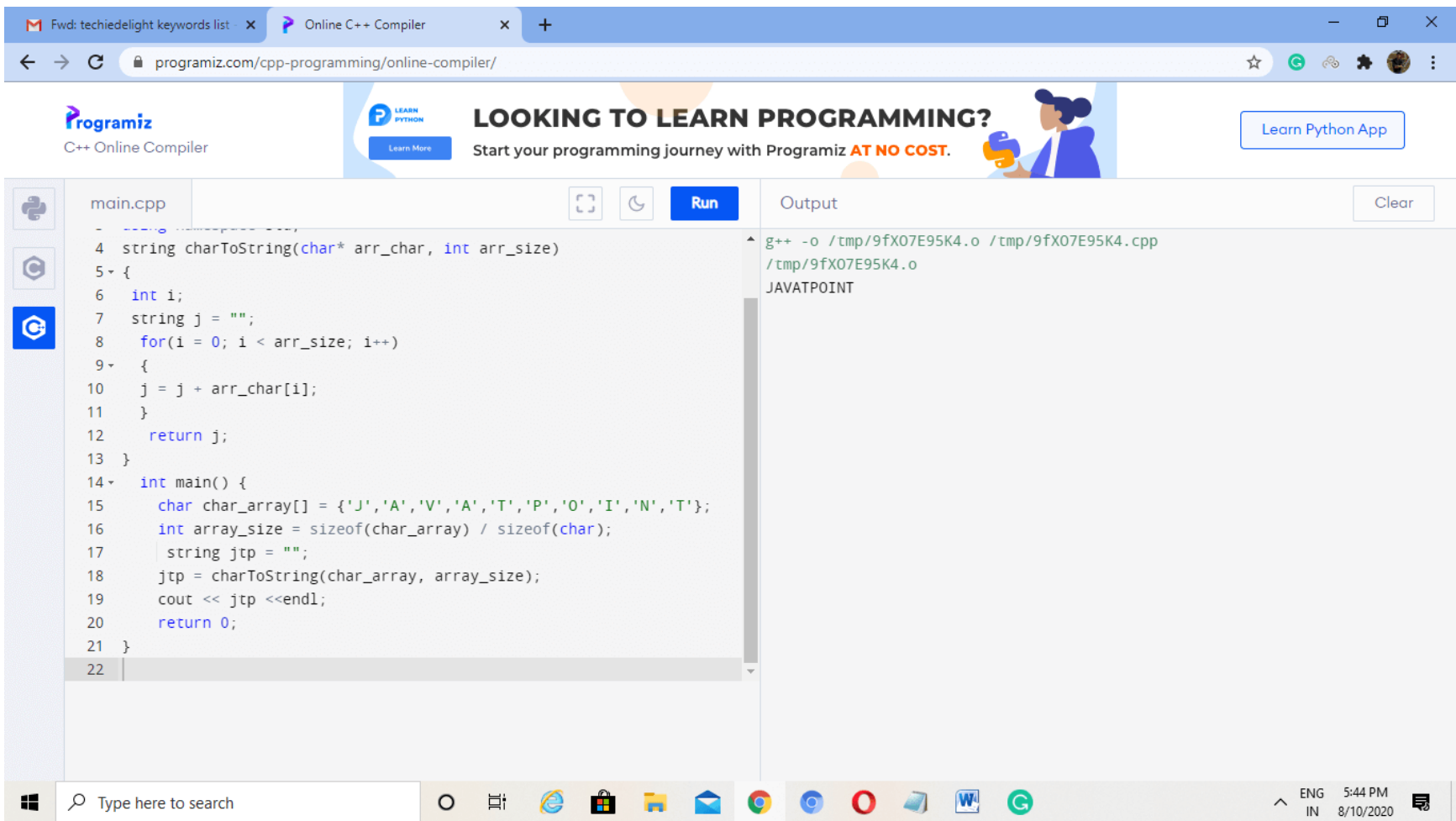
To convert character to a string using custom functions, we have to create a custom function with two parameters. Inside the custom function, we have to declare two variables string and integer. Then we use the 'for' loop, where we assign 0 to an int variable, int variable's size to be less than array_size, and int variable's value increasing by one at each iteration. The function will return the string. For the primary function, we declare the character array and its size, then we pass character array and its size to the custom function. At last, we print the string variable which stores the returned value of the custom function.

Code -

```
1. #include <iostream>
2. using namespace std;
3. string charToString(char* arr_char, int arr_size)
4. {
5.     int i;
6.     string j = "";
7.     for(i = 0; i < arr_size; i++)
8.     {
9.         j = j + arr_char[i];
10.    }
11.    return j;
12. }
13. int main() {
14.     char char_array[] = {'J','A','V','A','T','P','O','I','N','T'};
15.     int array_size = sizeof(char_array) / sizeof(char);
16.     string jtp = "";
17.     jtp = charToString(char_array, array_size);
18.     cout << jtp << endl;
19.     return 0;
```

20. }

Output:

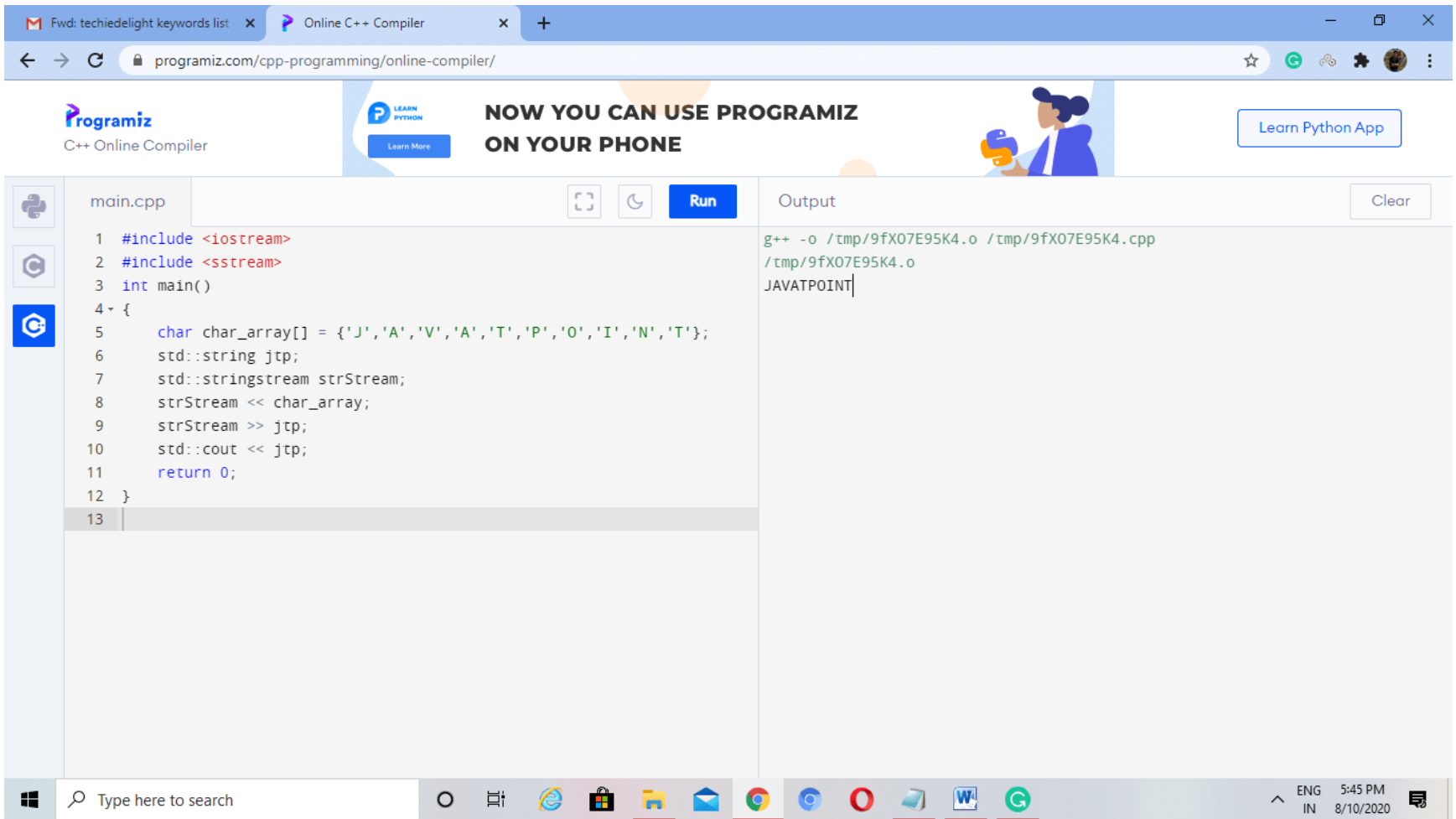


The last method to convert char to string is achieved by using **std::stringstream**. We use this function to insert the input character into a buffer and then take the character from the buffer as a string using **std::string**.

CODE -

1. `#include <iostream>`
2. `#include <sstream>`
3. `int main()`
4. `{`
5. `char char_array[] = {'J','A','V','A','T','P','O','I','N','T'};`
6. `std::string jtp;`
7. `std::stringstream strStream;`
8. `strStream << char_array;`
9. `strStream >> jtp;`
10. `std::cout << jtp;`
11. `return 0;`
12. `}`

Output:



Calculator Program in C++

A calculator is a portable device that helps to perform simple mathematical calculations in our daily lives such as **addition, subtraction, division, multiplication**, etc. Some of the scientific calculators are used to perform complex calculation more easily like square root, function, exponential operations, logarithm, trigonometric function and hyperbolic function, etc. In this section, we will create **calculator program in C++ using function and do-while loop**.



Using Function

Lets' create a calculator program in C++ using the function and Switch statement.

1. `#include<iostream.h>`
2. `#include<stdio.h>`
3. `#include<conio.h>`
4. `#include<math.h>`
5. `#include<stdlib.h>`
6. `void add();`
7. `void sub();`
8. `void multi();`
9. `void division();`
10. `void sqr();`


```

11. void srt();
12. void exit();
13. void main()
14. {
15. clrscr();
16. int opr;
17. // display different operation of the calculator
18. do
19. {
20. cout << "Select any operation from the C++ Calculator"
21.     "\n1 = Addition"
22.     "\n2 = Subtraction"
23.     "\n3 = Multiplication"
24.     "\n4 = Division"
25.     "\n5 = Square"
26.     "\n6 = Square Root"
27.     "\n7 = Exit"
28.     "\n\n Make a choice: ";
29.     cin >> opr;
30.
31.     switch (opr)
32.     {
33.         case 1:
34.             add(); // call add() function to find the Addition
35.             break;
36.         case 2:
37.             sub(); // call sub() function to find the subtraction
38.             break;
39.         case 3:
40.             multi(); // call multi() function to find the multiplication
41.             break;
42.         case 4:
43.             division(); // call division() function to find the division
44.             break;
45.         case 5:
46.             sqr(); // call sqr() function to find the square of a number
47.             break;
48.         case 6:
49.             srt(); // call srt() function to find the Square Root of the given number
50.             break;
51.         case 7:
52.             exit(0); // terminate the program
53.             break;
54.         default:
55.             cout << "Something is wrong..!!";
56.             break;
57.     }
58.     cout << "\n-----\n";
59. }while(opr != 7);
60. getch();
61. }
62.
63. void add()
64. {
65.     int n, sum = 0, i, number;
66.     cout << "How many numbers you want to add: ";
67.     cin >> n;

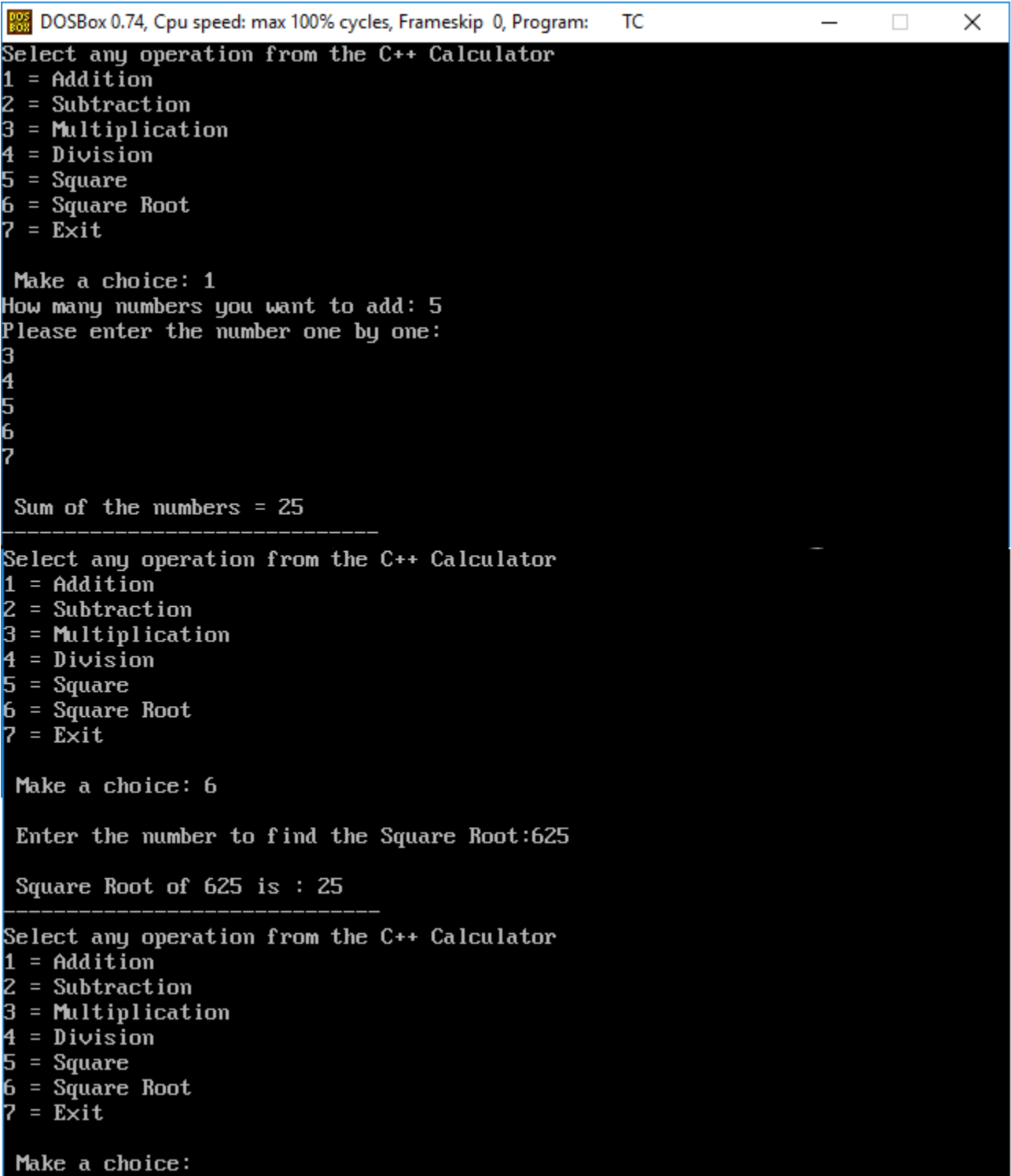
```



```
68. cout << "Please enter the number one by one: \n";
69. for (i = 1; i <= n; i++)
70. {
71. cin >> number;
72. sum = sum + number;
73. }
74. cout << "\n Sum of the numbers = " << sum;
75. }
76. void sub()
77. {
78. int num1, num2, z;
79. cout << " \n Enter the First number = ";
80. cin >> num1;
81. cout << "\n Enter the Second number = ";
82. cin >> num2;
83. z = num1 - num2;
84. cout << "\n Subtraction of the number = " << z;
85. }
86. void multi()
87. {
88. int num1, num2, mul;
89. cout << " \n Enter the First number = ";
90. cin >> num1;
91. cout << "\n Enter the Second number = ";
92. cin >> num2;
93. mul = num1 * num2;
94. cout << "\n Multiplication of two numbers = " << mul;
95. }
96. void division()
97. {
98. int num1, num2, div = 0;
99. cout << " \n Enter the First number = ";
100.     cin >> num1;
101.     cout << "\n Enter the Second number = ";
102.     cin >> num2;
103.     while ( num2 == 0)
104.     {
105.         cout << "\n Divisor cannot be zero"
106.         "\n Please enter the divisor once again: ";
107.         cin >> num2;
108.     }
109.     div = num1 / num2;
110.     cout << "\n Division of two numbers = " << div;
111. }
112. void sqr()
113. {
114.     int num1;
115.     float sq;
116.     cout << " \n Enter a number to find the Square: ";
117.     cin >> num1;
118.     sq = num1 * num1;
119.     cout << " \n Square of " << num1 << " is : " << sq;
120. }
121. void srt()
122. {
123.     float q;
124.     int num1;
```

```
125.     cout << "\n Enter the number to find the Square Root:";
126.     cin >> num1;
127.     q = sqrt(num1);
128.     cout << "\n Square Root of " << num1 << " is : " << q;
129.     }
```

Output:



Using do-while Loop

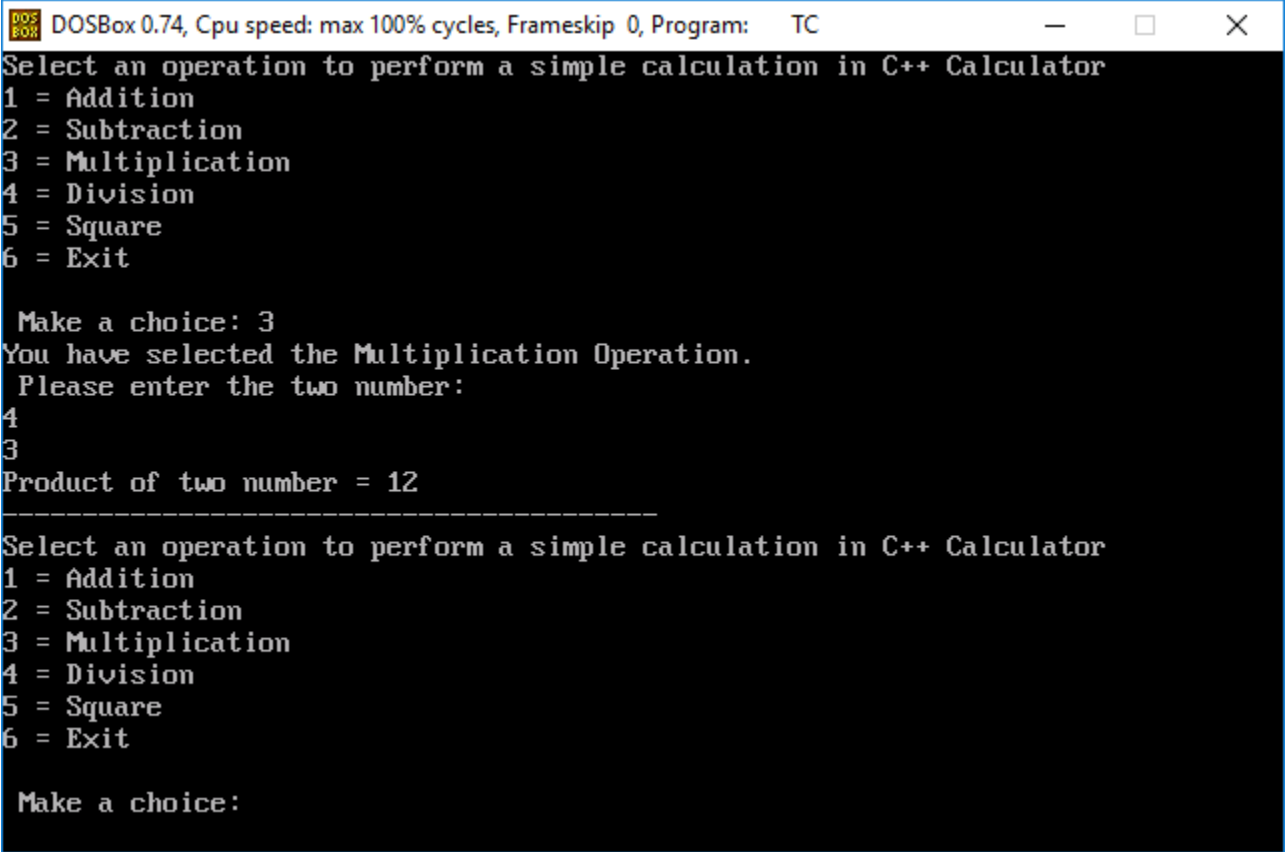
Write a Calculator Program in the C++ using the **do while** and **Switch** Statement.

```
1.  #include<iostream.h>
2.  #include<stdio.h>
3.  #include<conio.h>
4.  #include<stdlib.h>
5.  void main()
6.  {
7.  clrscr();
8.  int opr;
9.  int num1, num2, x;
10. // display different operation of the calculator
11. do
12. {
13. cout << "Select an operation to perform a simple calculation in C++ Calculator"
14.     "\n1 = Addition"
15.     "\n2 = Subtraction"
16.     "\n3 = Multiplication"
```

```
17.     "\n4 = Division"
18.     "\n5 = Square"
19.     "\n6 = Exit"
20.     "\n \n Make a choice: ";
21.     cin >> opr;
22.     switch (opr)
23.     {
24.         // for addition operation in calculator
25.         case 1:
26.             cout << "You have selected the Addition Operation.";
27.             cout << "\n Please enter the two number: \n";
28.             cin >> num1 >> num2;
29.             x = num1 + num2;
30.             cout << "Sum of two number = " << x;
31.             break;
32.         // for subtraction operation in calculator
33.         case 2:
34.             cout << "You have selected the Subtraction Operation.";
35.             cout << "\n Please enter the two number: \n";
36.             cin >> num1 >> num2;
37.             x = num1 - num2;
38.             cout << "Subtraction of two number = " << x;
39.             break;
40.         // for multiplication operation in calculator
41.         case 3:
42.             cout << "You have selected the Multiplication Operation.";
43.             cout << "\n Please enter the two number: \n";
44.             cin >> num1 >> num2;
45.             x = num1 * num2;
46.             cout << "Product of two number = " << x;
47.             break;
48.         // for division operation in calculator
49.         case 4:
50.             cout << "You have selected the Division Operation.";
51.             cout << "\n Please enter the two number; \n";
52.             cin >> num1 >> num2;
53.             // while loop checks for divisor whether it is zero
54.             while ( num2 == 0)
55.             {
56.                 cout << "\n Divisor cannot be zero"
57.                     "\n Please enter the divisor once again: ";
58.                 cin >> num2;
59.             }
60.             x = num1 / num2;
61.             cout << "\n Quotient = " << x;
62.             break;
63.         // to square a number in calculator
64.         case 5:
65.             cout << "You have selected the Square Operation.";
66.             cout << "\n Please enter any number: \n";
67.             cin >> num1;
68.             x = num1 * num1;
69.             cout << "Square is = " << x;
70.             break;
71.         case 6: exit(0); // terminate the program
72.         break;
73.         default: cout << "\n Something went wrong..!!";
```

```
74.     break;
75. }
76. cout << "\n----- \n";
77. } while(opr != 6);
78. getch();
79. }
```

Output:



```
DOSBox 0.74, Cpu speed: max 100% cycles, Frameskip 0, Program: TC
Select an operation to perform a simple calculation in C++ Calculator
1 = Addition
2 = Subtraction
3 = Multiplication
4 = Division
5 = Square
6 = Exit

Make a choice: 3
You have selected the Multiplication Operation.
Please enter the two number:
4
3
Product of two number = 12
-----
Select an operation to perform a simple calculation in C++ Calculator
1 = Addition
2 = Subtraction
3 = Multiplication
4 = Division
5 = Square
6 = Exit

Make a choice:
```

Program to convert infix to postfix expression in C++ using the Stack Data Structure

Infix expression

An infix expression is an expression in which operators (+, -, *, /) are written between the two operands. For example, consider the following expressions:

1. A + B
2. A + B - C
3. (A + B) + (C - D)

Here we have written '+' operator between the operands A and B, and the - operator in between the C and D operand.

Postfix Expression

The postfix operator also contains operator and operands. In the postfix expression, the operator is written after the operand. It is also known as **Reverse Polish Notation**. For example, consider the following expressions:

1. A B +
2. A B + C -
3. A B C * +
4. A B + C * D -

Algorithm to Convert Infix to Postfix Expression Using Stack

Following is the **algorithm** to convert infix expression into Reverse Polish notation.

1. Initialize the Stack.
2. Scan the operator from left to right in the infix expression.
3. If the leftmost character is an operand, set it as the current output to the Postfix string.
4. And if the scanned character is the operator and the Stack is empty or contains the '(', ')' symbol, push the operator into the Stack.

- 5. If the scanned operator has higher precedence than the existing **precedence** operator in the Stack or if the Stack is empty, put it on the Stack.
- 6. If the scanned operator has lower precedence than the existing operator in the Stack, pop all the Stack operators. After that, push the scanned operator into the Stack.
- 7. If the scanned character is a left bracket '(', push it into the Stack.
- 8. If we encountered right bracket ')', pop the Stack and print all output string character until '(' is encountered and discard both the bracket.
- 9. Repeat all steps from 2 to 8 until the infix expression is scanned.
- 10. Print the Stack output.
- 11. Pop and output all characters, including the operator, from the Stack until it is not empty.

Let's translate an infix expression into postfix expression in the stack:

Here, we have infix expression **((A * (B + D)/E) - F * (G + H / K))** to convert into its equivalent postfix expression:

Label No.	Symbol Scanned	Stack	Expression
1	((
2	(((
3	A	((A
4	*	((*	A
5	(((*(A
6	B	((*(AB
7	+	((*(+	AB
8	D	((*(+	ABD
9)	((*	ABD+
10	/	((*/	ABD+
11	E	((*/	ABD+E
12)	(ABD+E/*
13	-	(-	ABD+E/*
14	((-(ABD+E/*
15	F	(-(ABD+E/*F
16	*	(-(*	ABD+E/*F
17	((-(*(ABD+E/*F
18	G	(-(*(ABD+E/*FG
19	+	(-(*(+	ABD+E/*FG
20	H	(-(*(+	ABD+E/*FGH

21	/	(-(*(+/	ABD+E/*FGH
22	K	(-(*(+/	ABD+E/*FGHK
23)	(-(*	ABD+E/*FGHK/+
24)	(-	ABD+E/*FGHK/+*
25)		ABD+E/*FGHK/+*-

Program to convert infix expression to postfix expression

Let's create a C++ program that converts infix expression to the postfix expression

```
1. #include<iostream>
2. #include<stack>
3. using namespace std;
4. // defines the Boolean function for operator, operand, equalOrhigher precedence and the string conversion function.
5. bool IsOperator(char);
6. bool IsOperand(char);
7. bool eqlOrhigher(char, char);
8. string convert(string);
9.
10. int main()
11. {
12. string infix_expression, postfix_expression;
13. int ch;
14. do
15. {
16. cout << " Enter an infix expression: ";
17. cin >> infix_expression;
18. postfix_expression = convert(infix_expression);
19. cout << "\n Your Infix expression is: " << infix_expression;
20. cout << "\n Postfix expression is: " << postfix_expression;
21. cout << "\n \t Do you want to enter infix expression (1/ 0)?";
22. cin >> ch;
23. //cin.ignore();
24. } while(ch == 1);
25. return 0;
26. }
27.
28. // define the IsOperator() function to validate whether any symbol is operator.
29. /* If the symbol is operator, it returns true, otherwise false. */
30. bool IsOperator(char c)
31. {
32. if(c == '+' || c == '-' || c == '*' || c == '/' || c == '^' )
33. return true;
34. return false;
35. }
36.
37. // IsOperand() function is used to validate whether the character is operand.
38. bool IsOperand(char c)
39. {
40. if( c >= 'A' && c <= 'Z') /* Define the character in between A to Z. If not, it returns False.*/
41. return true;
42. if (c >= 'a' && c <= 'z') // Define the character in between a to z. If not, it returns False. */
43. return true;
44. if(c >= '0' && c <= '9') // Define the character in between 0 to 9. If not, it returns False. */
```

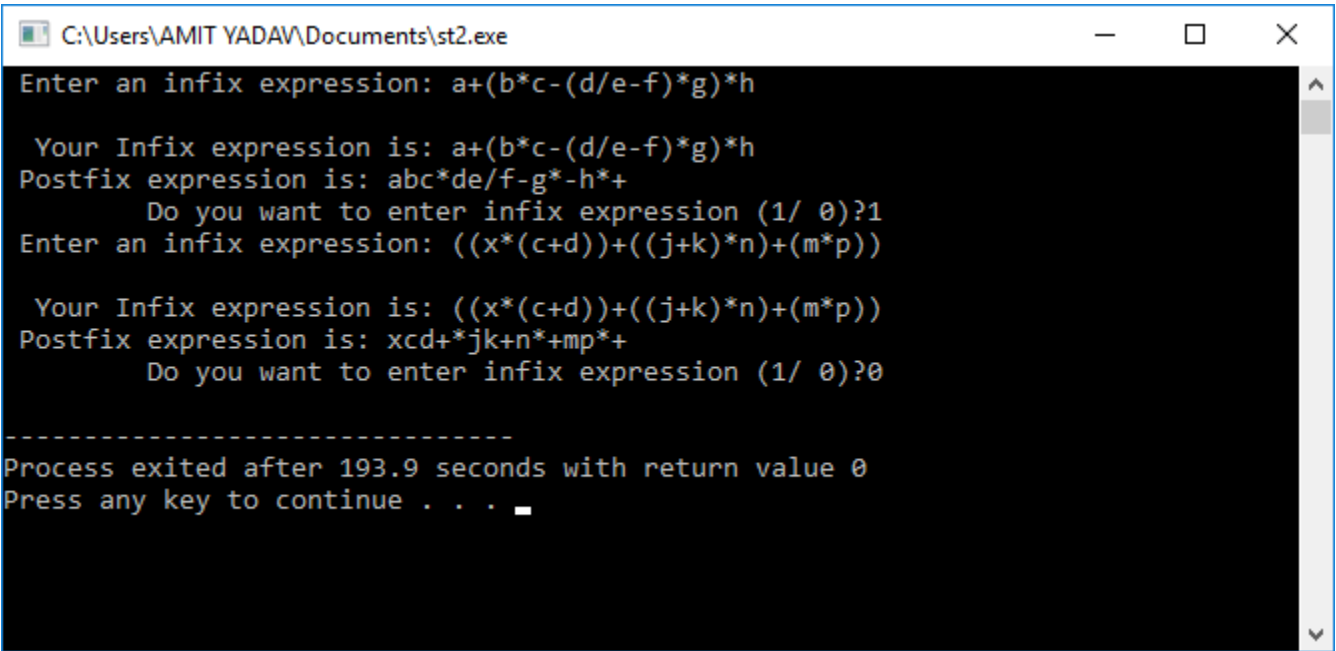
```

45. return true;
46. return false;
47. }
48. // here, precedence() function is used to define the precedence to the operator.
49. int precedence(char op)
50. {
51. if(op == '+' || op == '-')          /* it defines the lowest precedence */
52. return 1;
53. if (op == '*' || op == '/')
54. return 2;
55. if(op == '^')                      /* exponent operator has the highest precedence */
56. return 3;
57. return 0;
58. }
59. /* The eqOrhigher() function is used to check the higher or equal precedence of the two operators in infix expression. */
60. bool eqOrhigher (char op1, char op2)
61. {
62. int p1 = precedence(op1);
63. int p2 = precedence(op2);
64. if (p1 == p2)
65. {
66. if (op1 == '^' )
67. return false;
68. return true;
69. }
70. return (p1>p2 ? true : false);
71. }
72.
73. /* string convert() function is used to convert the infix expression to the postfix expression of the Stack */
74. string convert(string infix)
75. {
76. stack <char> S;
77. string postfix = "";
78. char ch;
79.
80. S.push( '(' );
81. infix += ')';
82.
83. for(int i = 0; i<infix.length(); i++)
84. {
85. ch = infix[i];
86.
87. if(ch == ' ')
88. continue;
89. else if(ch == '(')
90. S.push(ch);
91. else if(IsOperand(ch))
92. postfix += ch;
93. else if(IsOperator(ch))
94. {
95. while(!S.empty() && eqOrhigher(S.top(), ch))
96. {
97. postfix += S.top();
98. S.pop();
99. }
100.     S.push(ch);

```

```
101.     }
102.     else if(ch == ')')
103.     {
104.         while(!S.empty() && S.top() != '(')
105.         {
106.             postfix += S.top();
107.             S.pop();
108.         }
109.         S.pop();
110.     }
111.     }
112.     return postfix;
113.     }
```

Output:



C++ program to merge two unsorted arrays

In this article, we will write a program to merge two unsorted arrays. The output is the sorted array in ascending order.

Input : a[] = {10, 5, 15}

b[] = {20, 3, 2}

Output : The merged array in sorted order {2, 3, 5, 10, 15, 20}

Input : a[] = {1, 10, 5, 15}

b[] = {20, 0, 2}

Output : The merged array in sorted order {0, 1, 2, 5, 10, 15, 20}

Approach 1

The first approach that can be used here is to concatenate both the arrays and sort the concatenated array. We create a third array of the size of the first two arrays and then transfer all the elements of both the arrays into the resultant array. After the append operation, we sort the resultant array.

C code

```
1. #include <bits/stdc++.h>
2. using namespace std;
3.
4. void sortedMerge(int a[], int b[], int res[],
5.     int n, int m) // Merge two arrays in unsorted manner
6. {
7.     // Concatenate two arrays
```



```

8.   int i = 0, j = 0, k = 0;
9.   while (i < n) { // iterate in first array
10.    res[k] = a[i]; // put each element in res array
11.    i += 1;
12.    k += 1;
13. }
14. while (j < m) { // iterate in the second array
15.    res[k] = b[j]; // put each element in res array
16.    j += 1;
17.    k += 1;
18. }
19.
20. sort(res, res + n + m); // sort the res array to get the sorted Concatenate
21.}
22.
23.int main()
24.{
25.   int a[] = { 10, 5, 15, 22, 100 };
26.   int b[] = { 20, 3, 2, 12, 1, 7 };
27.   int n = sizeof(a) / sizeof(a[0]); // find the size of array a
28.   int m = sizeof(b) / sizeof(b[0]); // find the size of array b
29.
30.   int res[n + m]; // create res array to Concatenate both the array
31.   sortedMerge(a, b, res, n, m); // call function to append and sort
32.
33.   cout << "The sorted array is: ";
34.   for (int i = 0; i < n + m; i++)
35.       cout << " " << res[i];
36.   cout << "\n";
37.
38.   return 0;
39.}

```

Output

```
The sorted array is:  1 2 3 5 7 10 12 15 20 22 100
```

Time complexity

$O((n+m)\log(n+m))$, where **n** and **m** are the sizes of the arrays

Space complexity

$O(n+m)$

Approach 2

The above approach can be optimised when we use the idea of first sorting then merging it to the third array. Sort both the arrays separately and merge them into the resultant array.

C code

```

1. // CPP program to merge two unsorted lists
2. // in sorted order
3. #include <bits/stdc++.h>
4. using namespace std;
5. void sortedMerge(int a[], int b[], int res[],
6.   int n, int m) // Merge two arrays in unsorted manner
7.
8. {
9.   sort(a, a + n); // Sort the array a

```

```

10. sort(b, b + m); // sort the array b
11.
12.
13. int i = 0, j = 0, k = 0;
14. while (i < n && j < m) { // After the array are sorted compare and merge to third array
15.     if (a[i] <= b[j]) { // if element of a is less than b
16.         res[k] = a[i]; // put element of a into the res and increment i
17.         i += 1;
18.         k += 1;
19.     } else {
20.         res[k] = b[j]; // otherwise put the element of b into the res array and increment j
21.         j += 1;
22.         k += 1;
23.     }
24. }
25. while (i < n) { // If array a elements are left in the array put in res
26.     res[k] = a[i];
27.     i += 1;
28.     k += 1;
29. }
30. while (j < m) { // If array a elements are left in the array put in res
31.     res[k] = b[j];
32.     j += 1;
33.     k += 1;
34. }
35.}
36.
37. int main()
38. {
39.     int a[] = { 10, 5, 15, 22, 100 };
40.     int b[] = { 20, 3, 2, 12, 1, 7 };
41.     int n = sizeof(a) / sizeof(a[0]); // find the size of array a
42.     int m = sizeof(b) / sizeof(b[0]); // find the size of array b
43.
44.     int res[n + m]; // create res array to Concatenate both the array
45.     sortedMerge(a, b, res, n, m); // call function to append and sort
46.
47.     cout << "The sorted array is: ";
48.     for (int i = 0; i < n + m; i++)
49.         cout << " " << res[i];
50.     cout << "\n";
51.
52.     return 0;
53.}

```

Output

```
The sorted array is: 1 2 3 5 7 10 12 15 20 22 100
```

Time Complexity:

$O(n \log n + m \log m + (n + m))$ // which is far better than approach 1

Space Complexity:

$O(n + m)$ It remains same as the approach 1

C++ coin change program

In this article, we will see the most asked interview problem. The coin change problem is a good example of a dynamic programming approach. Now let us understand the problem statement.

Problem statement

Given N and an array(say coins[]) that contains some numbers(coins in rupees). N is a coin, and the array contains various coins. The task is to make the change of N using the coins of the array. Make a change in such a way that a minimum number of coins are used.

Example

Let us take a input array coins[] = {10, 25, 5}, total coins = 3

We have N = 30.

The output is **two** as we can use one 25 rupee coin and a 5 rupee coin to make 30. (25 + 5 = 30)

Similarly, coins[] = {1, 9, 6, 5}, total coins = 4

N = 13

The output is **three** as we need two 6 rupees coins and one 1 rupee coin. (6 + 6 + 1 = 13)

Approach 1

The approach uses a recursive method. We start with N, and in each iteration, we divide the problem into smaller subproblems. This is done by taking every coin from the coins array and decrease it from N. The case where N comes to zero gives us the required solution. For an optimal answer, we return the minimum of all combinations.

C++ code

```
1. #include <bits/stdc++.h>
2. using namespace std;
3.
4.
5. int minCoins(int coins[], int total_coins, int N) // Function to return the minimum // coins needed
6. {
7.     if (N == 0) // If we have a combination then
8.         return 0;
9.
10.    int result = INT_MAX; // Currently result is initialised as INT_MAX
11.
12.    for (int i = 0; i < total_coins; i++) // run until availability of coins
13.    {
14.        if (coins[i] <= N) { // Add to the list of counting
15.            int sub_res = 1 + minCoins(coins, total_coins, N - coins[i]); // add 1 due to the coin inclusion
16.            // see if result can minimize
17.            if (sub_res < result)
18.                result = sub_res;
19.        }
20.    }
21.    return result;
22.}
23.
24.int main()
25.{
26.    int coins[] = { 10, 25, 5 };
27.    int sum = 30; // the money to convert
28.    int total_coins = 3; // total availability of coins
29.    cout << "Minmum coins needed are " << minCoins(coins, total_coins, sum);
30.}
```

Output

```
Minimum coins needed are 2
```

The above solution does not work for more significant inputs. In this case, the recursive approach fails.

Now let us look at an optimized approach.

Approach 2

This approach uses the idea of bottom-up dynamic programming.

Since subproblems are computed, again and again, there is a condition of overlapping subproblems. The problem of recomputation can be solved using the property of memoization, i.e., create a `dp[]` array that stores the computed values at a particular stage. At each stage calculate all the possible combinations. After all the combinations, the last value in the `dp[]` array gives us the answer.

C++ code

```
1. #include <bits/stdc++.h>
2. using namespace std;
3.
4. int coins[] = { 10, 25, 5 }; // coins array
5. int dp[1000] = { 0 }; // array for memoisation
6. int minCoins(int N, int M) // N is the sum , M is total_coins
7. {
8.
9.     for (int i = 0; i <= N; i++)
10.         dp[i] = INT_MAX; // Initialise all dp[] value to INT_MAX
11.
12.     dp[0] = 0; // Base case if sum becomes zero min rupees = 0
13.
14.     //Iterating in the outer loop for possible values of sum between 1 to N
15.     //Since our final solution for sum = N might depend upon any of these values
16.     for (int i = 1; i <= N; i++) {
17.         //Inner loop denotes the index of coin array.
18.         //For each value of sum, to obtain the optimal solution.
19.         for (int j = 0; j < M; j++) {
20.             //i ?> sum
21.             //j ?> next coin index
22.             //If we can include this coin in our solution
23.             if (coins[j] <= i) {
24.                 //Solution might include the newly included coin
25.                 dp[i] = min(dp[i], 1 + dp[i - coins[j]]);
26.             }
27.         }
28.     }
29.     return dp[N];
30. }
31.
32. int main()
33. {
34.
35.     int sum = 30; // the money to convert
36.     int total_coins = 3; // total availability of coins
37.     cout << "Minimum coins needed are " << minCoins(sum, total_coins);
38. }
```

Output

```
Minimum coins needed are 2
```

The time complexity of the code is $O(\text{total_coins} + \text{sum})$. This is a much-optimized approach because of storing the results at each stage and utilizing it on the other iterations.

C++ program to add two complex numbers using class

In this article, we will write a program to add two complex numbers **(*a1* + *ib1*)** and **(*a2* + *ib2*)** using class.

For example

Input: 4 + i5 and 8 + i9

Here a1= 4 and a2 = 8. On adding a1 and a2, we get (8 + 4) = 12

Further, b1 = 5 and b2 = 9. On adding b1 and b2, we get (5 + 9) = 14

Output: 9 + i14

Input: 2 + i7 and 10 + i6

Here a1= 2 and a2 = 10. On adding a1 and a2, we get (2 + 10) = 12

Further, b1 = 7 and b2 =6. On adding b1 and b2, we get (7 + 6) = 13

Output: 12 + i13

Class construction

At first, we will create a class for complex numbers. The observation shows that there is a real number(a1) and an imaginary number(b1) in a complex number.

We need two data members to represent the complex numbers.

Below is the class structure:

1. **class** Complex
2. {
3. **public:**
4. **int** real; // To store real part of complex number
5. **int** imaginary; // To store imaginary part of complex number
6. }

Constructors

Two constructors will be made to initialize the data members of the class Complex.

- **Non - Parameterized constructor**

We need a non - parameterized constructor to initialize the default values of the data members to zero.

Below is the structure of non-parameterized constructor

1. Complex()
2. {
3. real = 0;
4. imaginary = 0;
5. }

- **Parameterized constructor**

We need a non-parameterized constructor to initialize the data members to the value passed by the main function.

Below is the structure of parameterized constructor

1. Complex(**int** r, **int** i)
2. {
3. real = r; // r is initialized during object creation
4. imaginary = i; // i is initialized during object creation

- 5.
6. }

Algorithm

For adding two complex numbers, we will make two objects of the Complex class and initialize them with the values. After that, we will make a third object that will store the result.

C++ code

```
1. // C++ program to Add two complex numbers
2. #include <bits/stdc++.h>
3. using namespace std;
4.
5. class Complex {
6. public:
7.     int real; // To store real part of complex number
8.     int imaginary; // To store imaginary part of complex number
9.
10.    Complex()
11.    {
12.        // Initial values are zero
13.        real = 0;
14.        imaginary = 0;
15.    }
16.    Complex(int r, int i)
17.    {
18.        real = r; // r is initialized during object creation
19.        imaginary = i; // i is initialized during object creation
20.    }
21.
22.    Complex addComplexNumber(Complex C1, Complex C2)
23.    {
24.
25.        Complex res; // result object of complex class
26.
27.        // adding real part of complex numbers
28.        res.real = C1.real + C2.real;
29.
30.        // adding Imaginary part of complex numbers
31.        res.imaginary = C1.imaginary + C2.imaginary;
32.
33.        // returning the sum
34.        return res;
35.    }
36. };
37.
38. // Main Class
39. int main()
40. {
41.
42.    // First Complex number
43.    Complex C1(4, 5);
44.
45.    // printing first complex number
46.    cout << "Complex number 1 : " << C1.real
47.        << " + i" << C1.imaginary << endl;
48.
```

```

49. // Second Complex number
50. Complex C2(8, 9);
51.
52. // printing second complex number
53. cout << "Complex number 2 : " << C2.real
54.     << " + i" << C2.imaginary << endl;
55.
56. // for Storing the sum
57. Complex C3;
58.
59. // calling addComplexNumber() method
60. C3 = C3.addComplexNumber(C1, C2);
61.
62. // printing the sum
63. cout << "Sum of complex number : "
64.     << C3.real << " + i"
65.     << C3.imaginary;
66.
67. cout << endl
68.     << endl;
69. // Test for second input
70. // First Complex number
71. Complex A(2, 7);
72.
73. // printing first complex number
74. cout << "Complex number 1 : " << A.real
75.     << " + i" << A.imaginary << endl;
76.
77. // Second Complex number
78. Complex B(10, 6);
79.
80. // printing second complex number
81. cout << "Complex number 2 : " << B.real
82.     << " + i" << B.imaginary << endl;
83.
84. // for Storing the sum
85. Complex C;
86.
87. // calling addComplexNumber() method
88. C = C.addComplexNumber(A, B);
89.
90. // printing the sum
91. cout << "Sum of complex number : "
92.     << C.real << " + i"
93.     << C.imaginary;
94. }

```

95. Output

```

96. Complex number 1 : 4 + i5
97. Complex number 2 : 8 + i9
98. Sum of complex number : 12 + i14
99.
100. Complex number 1 : 2 + i7
101. Complex number 2 : 10 + i6
102. Sum of complex number : 12 + i13
103.
104. Time complexity
105. The time complexity is O(1) as we have to call the function addComplexNumber() to add the two complex
    numbers.

```

C++ program to find the GCD of two numbers

In this tutorial, we will write a c++ program to find the GCD of two numbers.

GCD (Greatest common divisor) is also known as HCF (Highest Common Factor).

For example

$$36 = 2 * 2 * 3 * 3$$

$$60 = 2 * 2 * 3 * 5$$

The highest common factor of the two numbers is 2, 2, and 3.

So, the HCF of two numbers is $2 * 2 * 3 = 12$

$$20 = 2 * 2 * 5$$

$$28 = 2 * 2 * 7$$

The highest common factor of the two numbers is 2 and 2.

So, the HCF of two numbers is $2 * 2 = 4$.

Approach 1

The problem can be solved by finding all the prime factors of the two numbers and then find common factors and return their product.

- Find the factors of the first number
- Find the factors of the second number
- Find common factors of both numbers
- Multiply them to get GCD

C++ code

```
1. #include <bits/stdc++.h>
2. #define MAXFACTORS 1024 // Let us consider max factors can be 1024
3. using namespace std;
4.
5. typedef struct
6. {
7.
8.     int size;
9.     int factor[MAXFACTORS + 1];
10.    int exponent[MAXFACTORS + 1];
11.
12. } FACTORIZATION;
13.
14. // Function to find the factorization of M and N
15. void FindFactorization(int x, FACTORIZATION* factorization)
16. {
17.     int i, j = 1;
18.     int n = x, c = 0;
19.     int k = 1;
20.     factorization->factor[0] = 1;
21.     factorization->exponent[0] = 1;
22.
23.     for (i = 2; i <= n; i++) {
24.         c = 0;
25.
26.         while (n % i == 0) {
27.             c++;
28.
29.             // factorization->factor[j]=i;
```



```

30.     n = n / i;
31.     // j++;
32. }
33.
34.     if (c > 0) {
35.         factorization->exponent[k] = c;
36.         factorization->factor[k] = i;
37.         k++;
38.     }
39. }
40.
41. factorization->size = k - 1;
42. }
43.
44.
45.
46. // function to find the gcd using Middle School procedure
47. int gcdMiddleSchoolProcedure(int m, int n)
48. {
49.
50.     FACTORIZATION mFactorization, nFactorization;
51.
52.     int r, mi, ni, i, k, x = 1, j;
53.
54.     // Step 1.
55.     FindFactorization(m, &mFactorization);
56.
57.
58.     // Step 2.
59.     FindFactorization(n, &nFactorization);
60.
61.
62.     // Steps 3 and 4.
63.     // Procedure algorithm for computing the
64.     // greatest common divisor.
65.     int min;
66.     i = 1;
67.     j = 1;
68.     while (i <= mFactorization.size && j <= nFactorization.size) {
69.         if (mFactorization.factor[i] < nFactorization.factor[j])
70.             i++;
71.
72.         else if (nFactorization.factor[j] < mFactorization.factor[i])
73.             j++;
74.
75.         else /* if arr1[i] == arr2[j] */
76.         {
77.             min = mFactorization.exponent[i] > nFactorization.exponent[j]
78.                 ? nFactorization.exponent[j]
79.                 : mFactorization.exponent[i];
80.
81.             x = x * mFactorization.factor[i] * min;
82.             i++;
83.             j++;
84.         }
85.     }
86.

```

```

87.     return x;
88. }
89.
90.
91. int main()
92.
93. {
94.
95.     int m = 36, n = 60;
96.     cout << "GCD of " << m << " and " << n << " is "
97.         << gcdMiddleSchoolProcedure(m, n);
98.
99.     return (0);
100. }

```

Output

```
GCD of 36 and 60 is 12
```

Approach 2

Approach 1 can be solved in an efficient way using the Euclidean algorithm. This algorithm follows the idea that GCD does not change if the smaller number gets subtracted from the larger one.

C++ code

```

1. #include <iostream>
2. using namespace std;
3.
4. int gcd(int a, int b) // The function runs recursive in nature to return GCD
5. {
6.
7.     if (a == 0) // If a becomes zero
8.         return b; // b is the GCD
9.     if (b == 0) // If b becomes zero
10.        return a; // a is the GCD
11.
12.
13.    if (a == b) // The case of equal numbers
14.        return a; // return any one of them
15.
16.    // Apply case of subtraction
17.    if (a > b) // if a is greater subtract b
18.        return gcd(a-b, b);
19.    return gcd(a, b-a); // otherwise subtract a
20. }
21.
22.
23. int main()
24. {
25.     int a = 36, b = 60;
26.     cout << "GCD of " << a << " and " << b << " is " << gcd(a, b);
27.     return 0;
28. }

```

Output

```
GCD of 36 and 60 is 12
```

Approach 3

Approach two can be optimized further to use a modulo operator instead of subtraction.

C++ code

```
1. #include <iostream>
2. using namespace std;
3.
4. int gcd(int a, int b) // The function runs recursive in nature to return GCD
5. {
6.     if (b == 0) // if b becomes 0 return a
7.         return a;
8.     return gcd(b, a % b); // divide to a by b to make smaller number
9.
10.}
11.
12.
13.int main()
14.{
15.    int a = 60, b = 36;
16.    cout<<"GCD of "<<a<<" and "<<b<<" is "<<gcd(a, b);
17.    return 0;
18.}
```

Output

```
GCD of 60 and 36 is 12
```

C++ program to find greatest of four numbers

In this tutorial, we will write a C++ program to find the greatest of four numbers.

For example

a = 10, b = 50, c = 20, d = 25

The greatest number is b 50

a = 35, b = 50, c = 99, d = 2

The greatest number is c 99

Approach 1

The approach is the traditional way of searching for the greatest among four numbers. The if condition checks whether a is greater and then use if-else to check for b, another if-else to check for c, and the last else to print d as the greatest.

Algorithm

- START
- INPUT FOUR NUMBERS A, B, C, D
- IF A > B THEN
IF A > C THEN
IF A > D THEN
A IS THE GREATEST
ELSE
D IS THE GREATEST
- ELSE IF B > C THEN
IF B > D THEN
B IS THE GREATEST
ELSE
D IS THE GREATEST
- ELSE IF C > D THEN
C IS THE GREATEST

- ELSE
D IS THE GREATEST

C++ Code

```
1. #include <bits/stdc++.h>
2. using namespace std;
3. void find_greatest(int a, int b, int c, int d)
4. {
5.     if (a > b) {
6.         if (a > c) {
7.             if (a > d) {
8.                 cout << "a is greatest";
9.             }
10.        else {
11.            cout << "d is greatest";
12.        }
13.    }
14. }
15. else if (b > c) {
16.     if (b > d) {
17.         cout << "b is greatest";
18.     }
19.     else {
20.         cout << "d is greatest";
21.     }
22. }
23. else if (c > d) {
24.     cout << "c is greatest";
25. }
26. else {
27.     cout << "d is greatest";
28. }
29.}
30.
31. int main()
32. {
33.     int a = 10, b = 50, c = 20, d = 25;
34.     cout << "a=" << a << " b=" << b << " c=" << c << " d=" << d;
35.     cout << "\n";
36.     find_greatest(a, b, c, d);
37.     a = 35, b = 50, c = 99, d = 2;
38.     cout << "\n";
39.     cout << "a=" << a << " b=" << b << " c=" << c << " d=" << d;
40.     cout << "\n";
41.     find_greatest(a, b, c, d);
42.
43.     return 0;
44. }
```

Output

```
a=10 b=50 c=20 d=25
b is greatest
a=35 b=50 c=99 d=2
c is greatest
```

Approach 2

This approach uses the inbuilt max function.

Here is the syntax of max function

template constexpr const T& max (const T& a, const T& b);

Here, a and b are the numbers to be compared.

Return: Larger of the two values.

For example

std :: max(2,5) will return 5

So, to find out the maximum of 4 numbers, we can use chaining of a max function as follows -

int x = max(a, max(b, max(c, d)));

C++ code

```
1. #include <bits/stdc++.h>
2. using namespace std;
3. void find_greatest(int a, int b, int c, int d)
4. {
5.     int x = max(a, max(b, max(c, d)));
6.     if (x == a)
7.         cout << "a is greatest";
8.     if (x == b)
9.         cout << "b is greatest";
10.    if (x == c)
11.        cout << "c is greatest";
12.    if (x == d)
13.        cout << "d is greatest";
14.}
15.
16.int main()
17.{
18.    int a = 10, b = 50, c = 20, d = 25;
19.    cout << "a=" << a << " b=" << b << " c=" << c << " d=" << d;
20.    cout << "\n";
21.    find_greatest(a, b, c, d);
22.    a = 35, b = 50, c = 99, d = 2;
23.    cout << "\n";
24.    cout << "a=" << a << " b=" << b << " c=" << c << " d=" << d;
25.    cout << "\n";
26.    find_greatest(a, b, c, d);
27.
28.    return 0;
29.}
```

Output

```
a=10 b=50 c=20 d=25
b is greatest
a=35 b=50 c=99 d=2
c is greatest
```

Delete Operator in C++

A delete operator is used to deallocate memory space that is dynamically created using the new operator, calloc and malloc() function, etc., at the run time of a program in C++ language. In other words, a delete operator is used to release array and non-array (pointer) objects from the heap, which the new operator dynamically allocates to put variables on heap memory. We can use either the delete operator or delete [] operator in our program to delete the deallocated space. A delete operator has a void return type, and hence, it does not return a value.



Syntax of delete operator

We can delete a specific element or variable using the delete operator, as shown:

1. **delete** pointer_variable;
2. *// delete ptr; It deallocates memory for one element*

Similarly, we can delete the block of allocated memory space using the delete [] operator.

1. **delete** [] pointer_variable;
2. *// delete [] ptr; It deallocate for an array*

Program to deallocate memory location of each variable using the delete operator

Let's consider an example to delete the allocated memory space of each variable from the heap memory using the delete operator.

Program1.cpp

```
1. #include <iostream>
2. using namespace std;
3. int main ()
4. {
5.     // declaration of variables
6.     int *ptr1, *ptr2, sum;
7.
8.     // allocated memory space using new operator
9.     ptr1 = new int;
10.    ptr2 = new int;
11.
12.    cout << " Enter first number: ";
13.    cin >> *ptr1;
14.    cout << " Enter second number: ";
15.    cin >> *ptr2;
16.    sum = *ptr1 + *ptr2;
17.    cout << " Sum of pointer variables = " << sum;
18.
19.    // delete pointer variables
20.    delete ptr1;
21.    delete ptr2;
22.    return 0;
23.}
```

Output

```
Enter first number: 5
Enter second number: 8
Sum of pointer variables = 13
```

Program to delete the array object using the delete [] operator

Let's create a program to delete the dynamically created memory space for an array object using the delete [] operator in C++.

Program2.cpp

```
1. #include <iostream>
2. using namespace std;
3. int main ()
4. {
5.     // declaration of variables
6.     int *arr, max_num, i;
7.
8.     cout << " Enter total number of elements to be entered : ";
9.     cin >> max_num;
10.
11.    // use new operator to declare array memory at run time
12.    arr = new int [max_num];
13.
14.    cout << " Enter the numbers: \n";
15.    for (i = 0; i < max_num; i++) // input array from user
16.    {
17.        cout << " Number " << i+1 << " is ";
18.        cin >> arr[i];
19.    }
20.
21.    cout << " Numbers are : ";
22.    for (i = 0; i < max_num; i++)
23.    {
24.        cout << arr[i] << "\t";
25.    }
26.
27.    // use delete operator to deallocate dynamic memory
28.    delete [] arr;
29.    return 0;
30. }
```

Output

```
Enter total number of elements to be entered : 7
Enter the numbers:
Number 1 is 45
Number 2 is 600
Number 3 is 78
Number 4 is 93
Number 5 is 29
Number 6 is 128
Number 7 is 32
Numbers are : 45      600      78      93      29      128      32
```

Program to delete NULL pointer using delete operator

Let's consider a program to delete NULL pointer using the delete operator in C++ programming language.

Program3.cpp

```
1. #include <iostream>
2. using namespace std;
3. int main ()
4. {
5.     // initialize the integer pointer as NULL
6.     int *ptr = NULL;
7.
8.     // delete the ptr variable
```

```
9. delete ptr;
10. cout << " The NULL pointer is deleted.";
11. return 0;
12. }
```

Output

```
The NULL pointer is deleted.
```

Delete a pointer with or without value using the delete operator

Let's consider an example to delete a pointer variable with or without a value using the delete operator in C++.

Program4.cpp

```
1. #include <iostream>
2. using namespace std;
3. int main ()
4. {
5. // Use new operator to create dynamic memory
6. int *ptr = new int;
7.
8. // Use new operator to dynamic memory space for an array
9. int *ptr2 = new int (10);
10. cout << " The value of ptr is: "<< *ptr << " \n ";
11. cout << " The value of ptr2 is: "<< *ptr2 << " \n ";
12.
13. // use delete keyword to delete the value stored in *ptr and *ptr2
14. delete ptr;
15. delete ptr2;
16. return 0;
17. }
```

Output

```
The value of ptr is: 1415071048
The value of ptr2 is: 10
```

Program to allocate dynamic memory using the malloc function and then delete using the delete operator

Let's consider an example of creating dynamic memory using the malloc function and then using the delete operator to delete allocated memory in the C++ programming language.

Program6.cpp

```
1. #include <iostream>
2. using namespace std;
3. int main ()
4. {
5. // create a dynamic memory using malloc() function
6. char *str = (char *) malloc (sizeof (char));
7. cout << " Dynamic memory is deleted using the delete operator. " << endl;
8. delete str; // use delete operator to delete the referencing pointer
9. return 0;
10. }
```

Output

```
Dynamic memory is deleted using the delete operator.
```

Program to delete variables of user defined data types

Let's write a program to demonstrate the deletion of user defined object using the delete operator.

Program7.cpp


```

1. #include <bits/stdc++.h>
2. using namespace std;
3.
4. struct Ptr
5. {
6.     static void operator delete (void *ptr1, std::size_t sz)
7.     {
8.         cout << " Custom deletion of size " << sz << endl;
9.         delete (ptr1); // use delete() function
10.    }
11.
12.    static void operator delete[] (void *ptr1, std::size_t sz)
13.    {
14.        cout << " Custom deletion of size " << sz;
15.        delete (ptr1); // use ::operator delete() function
16.    }
17. };
18.
19. int main()
20. {
21.     Ptr *data = new Ptr; // create dynamic memory
22.     delete data; // delete specific variable
23.
24.     Ptr *data2 = new Ptr[20];
25.     delete[] data2; // delete block of memory
26. }

```

Output

```

Custom deletion of size 1
Custom deletion of size 24

```

Program to delete a void pointer using the delete operator

Let's create a program to release the memory space of the void pointer using the delete operator in C++.

Program8.cpp

```

1. #include <iostream>
2. using namespace std;
3. int main ()
4. {
5.     // declare a void pointer
6.     void *str;
7.     cout << " The void pointer is deleted using the delete operator. " << endl;
8.     // delete the void pointer reference
9.     return 0;
10. }

```

Output

```

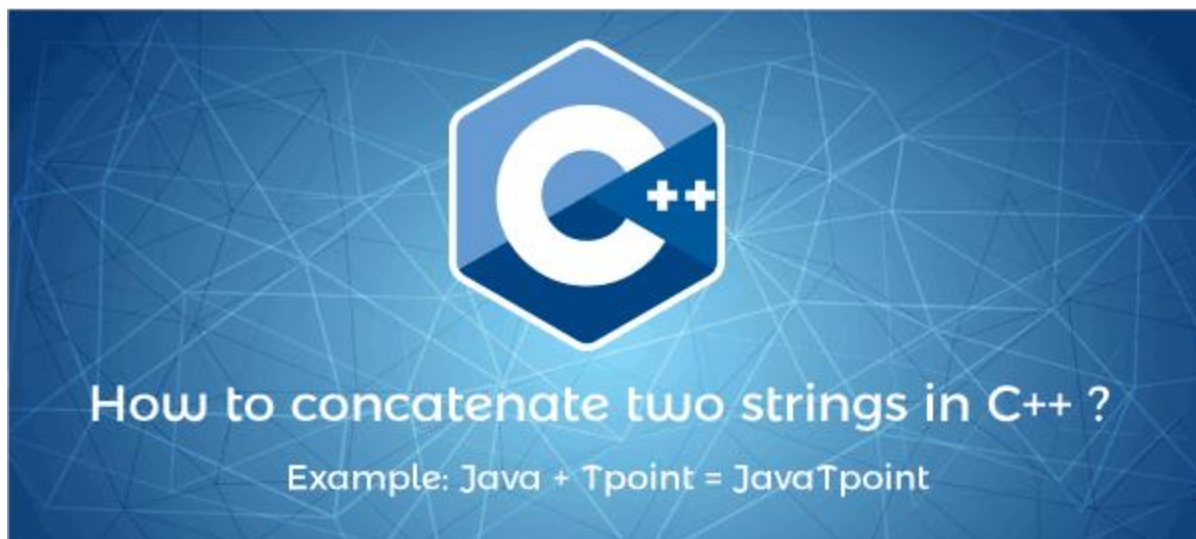
The void pointer is deleted using the delete operator.

```

How to concatenate two strings in c++?

This section will discuss the concatenation of two or more strings in the C++ programming language. The concatenation of the string means the group of characters that combines two more strings to return a concatenated single string. While concatenating the strings, the second string is added at the end of the first string to make a single string.

For example, we have two strings, 'Java' and 'Tpoint', and we want to concatenate to make a single string as Java + Tpoint = JavaTpoint.



Let's discuss the different ways to concatenate the given string in the C++ programming language.

1. Concatenate two strings using for loop
2. Concatenate two strings using while loop
3. Concatenate two strings using the + operator
4. Concatenate two strings using the strcat() function
5. Concatenate two strings using the append() function
6. Concatenate two strings using inheritance
7. Concatenate two strings using friend function and strcat() function

Program to concatenate two strings using for loop

Let's consider an example to combine two strings using for loop in the C++ programming.

Program.cpp

```
1. #include <iostream>
2. using namespace std;
3. int main ()
4. {
5.     string str1, str2, result; // declare string variables
6.     int i;
7.     cout << " Enter the first string: ";
8.     cin >> str1; // take string
9.     cout << " Enter the second string: ";
10.    cin >> str2; // take second string
11.    // use for loop to enter the characters of the str1 into result string
12.    for ( i = 0; i < str1.size(); i++)
13.    {
14.        result = result + str1[i]; // add character of the str1 into result
15.    }
16.
17.    // use for loop to enter the characters of the str2 into result string
18.    for ( i = 0; i < str2.size(); i++)
19.    {
20.        result = result + str2[i]; // add character of the str2 into result
21.    }
22.    cout << " The Concatenation of the string " << str1 << " and " << str2 << " is " << result;
23.    return 0;
24. }
```

Output

```
Enter the first string: Java
Enter the second string: Tpoint
The Concatenation of the string Java and Tpoint is JavaTpoint
```

Program to concatenate two strings using while loop

Let's consider an example to combine two strings using a while loop in C++ programming.

Program2.cpp

```
1. #include <iostream>
2. using namespace std;
3. int main ()
4. {
5.     // declare and initialize the string
6.     char str1[100] = " We Love";
7.     char str2[100] = " C++ Programming Language";
8.     int i, j; // declare variable
9.
10.    cout << " The first string is: " << str1 << endl;
11.    cout << " The second string is: " << str2 << endl;
12.    for (i = 0; str1[i] != '\0'; i++);
13.    j = 0; // initialize j with 0;
14.
15.    // use while loop that insert the str2 characters in str1
16.    while (str2[j] != '\0') // check str2 is not equal to null
17.    {
18.        str1[i] = str2[j]; // assign the character of str2 to str1
19.        i++;
20.        j++;
21.    }
22.    str1[i] = '\0';
23.    cout << " The concatenated string is: " << str1;
24.    return 0;
25.}
```

Output

```
The first string is:  We Love
The second string is:  C++ Programming Language
The concatenated string is:  We Love C++ Programming Language
```

Program to concatenate two strings using the + operator in C++

+ Operator: It is an arithmetic '+' operator that simply adds two strings to return a new concatenated string.

Let's consider an example to combine two strings using the + operator in C++ programming.

Program3.cpp

```
1. #include <iostream>
2. using namespace std;
3. int main ()
4. {
5.     string str1, str2; // declare string variables
6.
7.     cout << " Enter the first string: ";
8.     cin >> str1;
9.
10.    cout << "\n Enter the second string: ";
11.    cin >> str2;
12.
13.    // use '+' operator to concatenate the str1 and str2
```

```
14. string result = str1 + str2;
15. cout << " The concatenated string " << str1 << " and " << str2 << " is: " << result;
16. return 0;
17.}
```

Output

```
Enter the first string: Java
Enter the second string: Tpoint
The concatenated string Java and Tpoint is: JavaTpoint
```

Program to concatenate two strings using the strcat() method

strcat() function: The strcat is an inbuilt function of the string class, which adds two character strings to return a concatenated string.

Syntax

```
1. strcat ( char *arr1, char *arr2)
```

There are two character arrays in the above syntax, arr1 and arr2, which passed inside the strcat() function to return a concatenated string.

Let's consider an example to combine two strings using strcat() function in the C++ programming.

Program4.cpp

```
1. #include <bits/stdc++.h>
2. #include <string>
3. using namespace std;
4. int main()
5. {
6.     // declare and initialize the string
7.     char str1[] = " We love";
8.     char str2[] = " C++ Programming";
9.
10.    cout << " String 1: " <<str1 <<endl;
11.    cout << " String 2: " <<str2 <<endl;
12.    // use the strcat() function to concatenate the string
13.    strcat(str1, str2);
14.
15.    cout << " The concatenated string is: " <<str1;
16.    return 0;
17.}
```

Output

```
String 1:  We love
String 2: C++ Programming
The concatenated string is:  We love C++ Programming
```

Program to concatenate two strings using the append function

append() function: An **append()** function is a predefined library function used to insert or add a second string at the end of the first string to return a single string.

Syntax

```
1. str1.append(str2);
```

In the above syntax, the str2 is a second string to pass in the append() function that inserts the str2 string at the end of the str1 string.

Let's consider an example to combine two strings using append() function in the C++ programming.

Program5.cpp

```
1. #include <iostream>
```

```

2.  using namespace std;
3.  int main ()
4.  {
5.  string str1, str2, result; // declare string variables
6.
7.  cout << " Enter the first string: ";
8.  cin >> str1; // take string
9.  cout << " Enter the second string: ";
10. cin >> str2; // take second string
11.
12. // use append() function to insert element at the end of the str1
13. str1.append(str2);
14. cout << " \n The concatenation of the string is: " << str1;
15. return 0;
16. }

```

Output

```

Enter the first string: Hello
Enter the second string: Friends!

The concatenation of the string is: HelloFriends!

```

Program to concatenate two string using the inheritance of the class

Let's consider an example to combine two strings using inheritance in the C++ programming.

Program6.cpp

```

1.  #include <iostream>
2.  #include <string>
3.  using namespace std;
4.
5.  // create base class
6.  class base
7.  {
8.      protected:
9.          virtual string concatenate(string &str1, string &str2) = 0;
10. };
11.
12. // create derive class to acquire features of base class
13. class derive: protected base {
14.     public:
15.         string concatenate (string &str1, string &str2) {
16.             string temp;
17.             temp = str1 + str2;
18.             return temp;
19.         }
20. };
21.
22. int main()
23. {
24.     // declare variable
25.     string str1, str2;
26.
27.     cout << " Enter first string: ";
28.     cin >> str1;
29.     cout << " \n Enter second string: ";
30.     cin >> str2;
31.
32.     // create string object

```

```
33.   derive obj;
34.
35.   // print string
36.   cout <<" \n The concatenated string is: " << obj.concatenate (str1, str2);
37.
38.   return 0;
39. }
```

Output

```
Enter first string: C++
Enter second string: Programming
The concatenated string is: C++Programming
```

Program to concatenate two strings using the friend function and strcat() function

Let's consider an example to combine two strings using friend function and strcat() function in the C++ programming.

Program7.cpp

```
1.  #include <iostream>
2.  #include <cstring>
3.  using namespace std;
4.
5.  class Base {
6.  private:
7.      char st[100], st2[100];
8.
9.  public:
10.
11.     void inp()
12.     {
13.         cout <<" Enter the first string: ";
14.         cin.getline (st, 100); // take a line of string with limit 100
15.
16.         cout <<" Enter the second string: ";
17.         cin.getline (st2, 100); // take a line of string with limit 100
18.
19.
20.
21.     friend void myfun(Base b);
22. };
23.
24. void myfun (Base b)
25. {
26.     strcat (b.st, b.st2); // pass parameter to concatenate
27.
28.     cout <<" The concatenated string: " <<b.st;
29. }
30.
31. int main()
32. {
33.
34.     Base b; // create b as an object
35.
36.     b.inp(); // b object call inp() function
37.     myfun(b); // pass b object to myfun() to print the concatenated string
38.
39.     return 0;
40. }
```

Output

```
Enter the first string: javatpoint
Enter the second string: .com
The concatenated string: javatpoint.com
```

Upcasting and Downcasting in C++

This section will discuss Upcasting and Downcasting with an example in the C++ programming language. When we convert one data type into another type, the process is called typecasting. But the, Upcasting and downcasting are the types of object typecasting. Suppose the Parent and Child class has two types of object, parent_obj and child_obj, which can be cast into the Parent to Child and Child to Parent using the Upcasting and Downcasting in C++ programming.



Upcasting

It is the process to create the derived class's pointer or reference from the base class's pointer or reference, and the process is called Upcasting. It means the upcasting used to convert the reference or pointer of the derived class to a base class. Upcasting is safe casting as compare to downcasting. It allows the public inheritance that implicitly cast the reference from one class to another without an explicit typecast. By default, the upcasting create is-a relationship between the base and derived classes.

1. Base *ptr = &derived_obj;

The derived class can inherit all the base class properties that include data members and the member function to execute the function using the derived class object, as we do with a base object.

Program to demonstrate the Upcasting in C++

Let's consider an example to convert the derived class's pointer to the base class's pointer in the C++ programming language.

Program1.cpp

```
1. #include <iostream>
2. using namespace std;
3. class Base
4. {
5.     public:
6.         void disp()
7.         {
8.             cout << " It is the Super function of the Base class ";
9.         }
10. };
11.
12. class derive : public Base
13. {
14.     public:
15.         void disp()
16.         {
17.             cout << "\n It is the derive class function ";
18.         }
19. }
```

```

19.
20. };
21.
22. int main ()
23. {
24.     // create base class pointer
25.     Base *ptr;
26.
27.     derive obj; // create object of derive class
28.     ptr = &obj; // assign the obj address to ptr variable
29.
30.     // create base class's reference
31.     Base &ref = obj;
32.     // Or
33.     // get disp() function using pointer variable
34.
35.     ptr->disp();
36.     return 0;
37. }

```

Output

```
It is the Super function of the Base class
```

Downcasting

The Downcasting is an opposite process to the upcasting, which converts the base class's pointer or reference to the derived class's pointer or reference. It manually cast the base class's object to the derived class's object, so we must specify the explicit typecast. The downcasting does not follow the **is- a** relation in most of the cases. It is not safe as upcasting. Furthermore, the derived class can add new functionality such as; new data members and class member's functions that use these data members. Still, these functionalities could not apply to the base class.

```
1. Derived *d_ptr = &b_obj;
```

Program to demonstrate the downcasting in C++

Let's create an example to downcast the base class's object to the derived class in the C++ programming language.

Program2.cpp

```

1. #include <iostream>
2. using namespace std;
3. class Parent
4. {
5.     public:
6.         void base()
7.         {
8.             cout << " It is the function of the Parent class " << endl;
9.         }
10. };
11.
12. class Child : public Parent
13. {
14.     public:
15.         void derive()
16.         {
17.             cout << " it is the function of the Child class " << endl;
18.         }
19. };
20.
21. int main ()

```



```

22. {
23.   Parent pobj; // create Parent's object
24.   Child *cobj; // create Child's object
25.
26.   // explicit type cast is required in downcasting
27.   cobj = (Child *) &pobj;
28.   cobj -> derive();
29.
30.   return 0;
31.}

```

Output

It is the function of the Child class

Program to demonstrate the upcasting and the downcasting in C++

Let's consider an example to use the downcasting and the upcasting in C++ to convert the base class to derive and the derived class's object to the base class.

Program3.cpp

```

1. #include <iostream>
2. using namespace std;
3.
4. class Parent {
5.     private:
6.         int id;
7.
8.     public:
9.         void showid ()
10.        {
11.            cout << " I am in the Parent class " << endl;
12.        }
13.};
14.
15. class Myson : public Parent {
16.     public:
17.         void disp ()
18.        {
19.            cout << " I am in the Myson class " << endl;
20.        }
21.};
22.
23. int main ( int argc, char * argv[])
24. {
25.     // create object of the Parent class
26.     Parent par_obj;
27.
28.     // create object of the Myson class
29.     Myson my_obj;
30.
31.     // upcast - here upcasting can be done implicitly
32.     Parent *ptr1 = &my_obj; // base class's reference the derive class's object
33.
34.     // downcast - here typecasting is done explicitly
35.     Myson *ptr2 = (Myson *) &par_obj;
36.
37.     // Upcasting is safe:
38.     ptr1->showid();

```

```
39. ptr2->showid();
40.
41.
42. // downcasting is unsafe:
43. ptr2->disp();
44.
45. getchar();
46. return 0;
47.
48. }
```

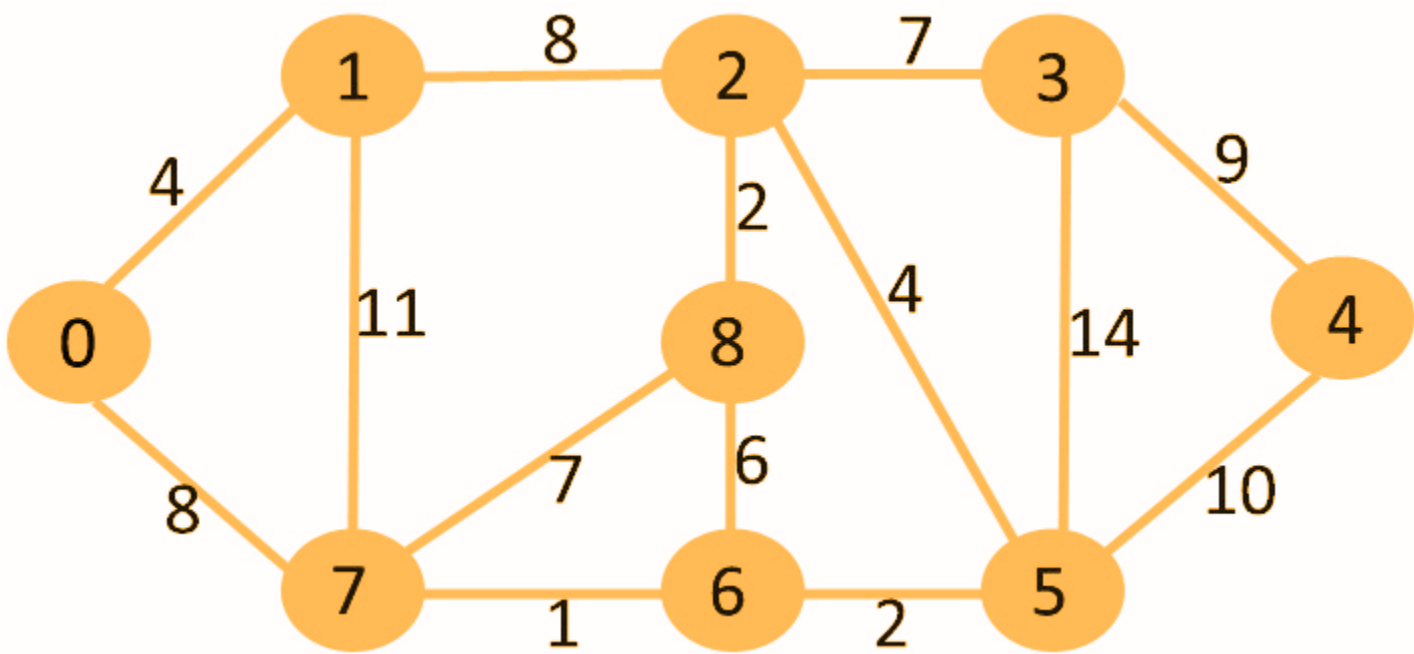
Output

```
I am in the Parent class
I am in the Parent class
I am in the Myson class
```

C++ Dijkstra Algorithm using the priority queue

In this article, we will see the implementation of the Dijkstra algorithm using the priority queue of C++ STL. Dijkstra algorithm is used to find the shortest path from the source to the destination in an undirected graph.

A graph having weight on the edges is given as below:



Let us consider a source vertex **0**, we have to find the shortest path from the source vertex to all the vertices in the graph.

Source vertex = 0

Vertex	Distance from source
0	0 Same source and destination
1	4 Directly goes to 1
2	12 Path: 0 ->1 -> 2 (8 + 4 = 12)
3	19 Path: 0 ->1 -> 2 -> 3 (8 + 4 + 7 = 19)
4	21 Path: 0 -> 7 -> 6 -> 5 -> 4 (8 + 1 + 2 + 10 = 21)

5	11 Path: 0 -> 7 -> 6 -> 5 (8 + 1 + 2 = 11)
6	9 Path: 0 -> 7 -> 6 (8 + 1 = 9)
7	8 Path: 0 -> 7
8	14 Path: 0 -> 1 -> 2 -> 8 (4 + 8 + 2 = 14)

Create graph structure

We will create a class Graph with data members as

- o **int v** - To store the number of vertices in the graph
- o List of pairs - To store the vertex and the weight associated with a particular vertex.

list> *adj;

Constructors:

We need a constructor to allocate the memory of the adjacency list.

1. Graph(**int** vertex)
2. {
3. **this**->V = vertex; // Allocate the number of vertices
4. adj = list<pair> [vertex]; // Allocate memory for adjacency list
5. }

How to add an edge to the graph?

The list of pairs created has two arguments. One will contain the vertex, and the other will contain the weight associated with it.

As the graph is bidirectional, we can add the same weight to the opposite vertex.

Code:

1. **void** addanEdge(**int** u, **int** v, **int** w)
2. {
3. adj[u].push_back(make_pair(v,w)); // add v to w
4. adj[v].push_back(make_pair(u,w)); add w to v
5. // To add a vertex with weight associated with it
6. }

Algorithm

1. Mark initial distance from the source is infinite.
2. Create an empty priority_queue PQ. Every item of PQ is a pair (weight, vertex). Weight (or distance) is used as the first item of pair as the first item is by default used to compare two pairs.
3. Insert source vertex into PQ and make its distance as 0.
4. Until the priority queue defined as PQ does not become empty. Perform the operations a and b.
 - a. Extract minimum distance vertex from PQ and let it be u.
 - b. Loop through all adjacent of u and do
 Following for every vertex v.
 // If there is a shorter path to v
 // through u.
 If dist[v] > dist[u] + weight(u, v) // distance of (v) > distance of (u) and weight from u to v

- a. Update distance of v, i.e., do
 $\text{dist}[v] = \text{dist}[u] + \text{weight}(u, v)$
 - b. Insert v into the pq (Even if v is already there)
5. Loop through the `dist[]` array to print the shortest paths from source to all the vertices.

C++ code

```

1. #include <bits/stdc++.h>
2. using namespace std;
3. #define INF 0x3f3f3f3f // The distance to other vertices is initialized as infinite
4. // iPair ==> Integer Pair
5. typedef pair<int, int> iPair;
6. class Graph // Graph structure
7. {
8.     int V; // No. of vertices in the graph
9.     list<pair<int, int>>* adj; // the list of pair to store vertex and its weight
10. public:
11.     // Constructor that accept number of vertices in graph
12.     Graph(int V) // allocate the vertex memory
13.     {
14.         this->V = V; // assign the vertex
15.         adj = new list<iPair>[V]; // allocate space for vertices
16.     }
17.     void addEdge(int u, int v, int w); // add edges in the graph
18.     // prints shortest path from s
19.     void shortestPathingraph(int s); // pass source vertex
20. };
21. void Graph::addEdge(int u, int v, int w) // add an edge
22. {
23.     adj[u].push_back(make_pair(v, w)); // make a pair of vertex and weight and // add it to the list
24.     adj[v].push_back(make_pair(u, w)); // add oppositely by making a pair of weight and vertex
25. }
26. // Calling function outside the Graph class
27. void Graph::shortestPathingraph(int src) // src is the source vertex
28. {
29.     // Create a priority queue to store vertices that
30.     // are being preprocessed.
31.     priority_queue<iPair, vector<iPair>, greater<iPair>> pq;
32.     vector<int> dist(V, INF); // All distance from source are infinite
33.     pq.push(make_pair(0, src)); // push source node into the queue
34.     dist[src] = 0; // distance of source will be always 0
35.     while (!pq.empty()) { // While queue is not empty
36.         // Extract the first minimum distance from the priority queue
37.         // vertex label is stored in second of pair (it
38.         // has to be done this way to keep the vertices
39.         // sorted distance
40.         int u = pq.top().second;
41.         pq.pop();
42.         // 'i' is used to get all adjacent vertices of a vertex
43.         list<pair<int, int>>::iterator i;
44.         for (i = adj[u].begin(); i != adj[u].end(); ++i) {
45.             // Get vertex label and weight of current adjacent
46.             // of u.
47.             int v = (*i).first;
48.             int weight = (*i).second;
49.

```

```

50.         // If there is shorted path to v through u.
51.         if (dist[v] > dist[u] + weight) {
52.             // Updating distance of v
53.             dist[v] = dist[u] + weight;
54.             pq.push(make_pair(dist[v], v));
55.         }
56.     }
57. }
58. printf("Vertex \tDistance from Source\n"); // Print the result
59. for (int i = 0; i < V; ++i)
60.     printf("%d \t\t %d\n", i, dist[i]); // The shortest distance from source
61. }
62. int main()
63. {
64.     int V = 9; // vertices in given graph are 9
65.     Graph g(V); // call Constructor by creating an object of graph
66.     g.addEdge(0, 1, 4); // add root node with neighbour vertex and weight
67.     g.addEdge(0, 7, 8);
68.     g.addEdge(1, 2, 8);
69.     g.addEdge(1, 7, 11);
70.     g.addEdge(2, 3, 7);
71.     g.addEdge(2, 8, 2);
72.     g.addEdge(2, 5, 4);
73.     g.addEdge(3, 4, 9);
74.     g.addEdge(3, 5, 14);
75.     g.addEdge(4, 5, 10);
76.     g.addEdge(5, 6, 2);
77.     g.addEdge(6, 7, 1);
78.     g.addEdge(6, 8, 6);
79.     g.addEdge(7, 8, 7);
80.     g.shortestPathingraph(0); // call the function to find shortest path of graph
81.     return 0; // end of main function()
82. }

```

Output

```

Vertex    Distance from Source
0          0
1          4
2         12
3         19
4         21
5         11
6          9
7          8
8         14

```

Constructor overloading in C++

Constructor is a member function of a class that is used to initialize the objects of the class. Constructors do not have any return type and are automatically called when the object is created.

Characteristics of constructors

- The name of the constructor is the same as the class name
- No return type is there for constructors
- Called automatically when the object is created
- Always placed in the public scope of the class
- If a constructor is not created, the default constructor is created automatically and initializes the data member as zero
- Declaration of constructor name is case-sensitive
- A constructor is not implicitly inherited

Types of constructors

There are three types of constructors -

- **Default constructor** - A default constructor is one that does not have function parameters. It is used to initialize data members with a value. The default constructor is called when the object is created.

Code

```
1. #include <iostream>
2. using namespace std;
3. class construct // create a class construct
4. {
5. public:
6.     int a, b; // initialise two data members
7.     construct() // this is how default constructor is created
8.     {
9.         // We can also assign both values to zero
10.        a = 10; // initialise a with some value
11.        b = 20; // initialise b with some value
12.    }
13.};
14. int main()
15. {
16.    construct c; // creating an object of construct calls default constructor
17.    cout << "a:" << c.a << endl
18.        << "b:" << c.b; // print a and b
19.    return 1;
20.}
```

Output

```
a:10
b:20
```

- **Parameterized constructor** - A non-parameterized constructor does have the constructor arguments and the value passed in the argument is initialized to its data members. Parameterized constructors are used in constructor overloading.

Code

```
1. #include <iostream>
2. using namespace std;
3. class Point // create point class
4. {
5. private:
6.     int x, y; // the two data members of class Point
7. public:
8.     Point(int x1, int y1) // create parameterised Constructor and initialise data member
9.     {
10.        x = x1; // x1 is now initialised to x
11.        y = y1; // y1 is now initialised to y
12.    }
13. int getX()
14. {
15.     return x; // to get the value of x
16. }
17. int getY()
18. {
19.     return y; // to get the value of y
20. }
```

```

21.};
22.int main()
23.{
24.    Point p1(10, 15); // created object for parameterised constructor
25.
26.    cout<< "p1.x = " << p1.getX() << ", p1.y = " << p1.getY(); // print x and y
27.
28.    return 0;
29.}

```

Output

```
p1.x = 10, p1.y = 15
```

Explanation

Create a class point with two data members x and y. A parameterized constructor is created with the points x1 and y1 as parameters and the value of x and y are assigned using x1 and y1. In the main function, we create a parameterized constructor with the values (10, 15). Using the getter functions, we get the values of the data members.

- **Copy constructor** - Copy constructor initializes an object using another object of the same class.

Syntax

class_name(constclassname&old_object).

Code

```

1. #include <iostream>
2. using namespace std;
3. class Point // create point class
4. {
5.     private:
6.     int x, y; // data members of the class
7.     public:
8.     Point(int x1, int y1)
9.     {
10.         x = x1;
11.         y = y1;
12.     } // parameterised constructor
13.
14.     // Copy constructor
15.     Point(const Point& p1) // initialisation according to syntax
16.     {
17.         x = p1.x;
18.         y = p1.y;
19.     }
20.     intgetX() { return x; } // return value of x
21.     intgetY() { return y; } // return value of y
22. };
23. int main()
24. {
25.     Point p1(10, 15); // call parameterised constructor
26.     Point p2 = p1; // call Copy constructor
27.     // use getter and setter to print x and y
28.     cout<< "p1.x = " << p1.getX() << ", p1.y = " << p1.getY();
29.     cout<< "\np2.x = " << p2.getX() << ", p2.y = " << p2.getY();
30.     return 0;
31. }

```

Output

```
p1.x = 10, p1.y = 15  
p2.x = 10, p2.y = 15
```

Constructor overloading in C++

As there is a concept of function overloading, similarly constructor overloading is applied. When we overload a constructor more than a purpose it is called constructor overloading.

The declaration is the same as the class name but as they are constructors, there is no return type.

The criteria to overload a constructor is to differ the number of arguments or the type of arguments.

Code

```
1. // C++ program to demonstrate constructor overloading  
2. #include <iostream>  
3. using namespace std;  
4. class Person { // create person class  
5. private:  
6. int age; // data member  
7. public:  
8. // 1. Constructor with no arguments  
9. Person()  
10. {  
11. age = 20; // when object is created the age will be 20  
12. }  
13. // 2. Constructor with an argument  
14. Person(int a)  
15. { // when parameterised Constructor is called with a value the  
16. // age passed will be initialised  
17. age = a;  
18. }  
19. intgetAge()  
20. { // getter to return the age  
21. return age;  
22. }  
23. };  
24. int main()  
25. {  
26. Person person1, person2(45); // called the object of person class in differnt way  
27.  
28. cout<< "Person1 Age = " << person1.getAge() <<endl;  
29. cout<< "Person2 Age = " << person2.getAge() <<endl;  
30. return 0;  
31. }
```

Output

```
Person1 Age = 20  
Person2 Age = 45
```

Explanation

In the above program, we have created a class **Person** with one data member(age). There are two constructors in the class that are overloaded. We have overloaded the second constructor by providing one argument and making it parameterized.

So, in the main function when the object person1 is created, it calls the non-parameterized constructor and when person2 is created, it calls the parameterized constructor and performs the required operation of initializing the age. Hence, when object person1 age is printed it gives 20 that was set by default and person2 age gives 45 as it was passed by the object to the parameterized constructor.

Default arguments in C++

In a function, arguments are defined as the values passed when a function is called. Values passed are the source, and the receiving function is the destination.

Now let us understand the concept of default arguments in detail.



Definition

A default argument is a value in the function declaration automatically assigned by the compiler if the calling function does not pass any value to that argument.

Characteristics for defining the default arguments

Following are the rules of declaring default arguments -

- The values passed in the default arguments are not constant. These values can be overwritten if the value is passed to the function. If not, the previously declared value retains.
- During the calling of function, the values are copied from left to right.
- All the values that will be given default value will be on the right.

Example

- `void function(int x, int y, int z = 0)`
Explanation - The above function is valid. Here z is the value that is predefined as a part of the default argument.
- `Void function(int x, int z = 0, int y)`
Explanation - The above function is invalid. Here z is the value defined in between, and it is not accepted.

Code

```
1. #include<iostream>
2. using namespace std;
3. int sum(int x, int y, int z=0, int w=0) // Here there are two values in the default arguments
4. { // Both z and w are initialised to zero
5.     return (x + y + z + w); // return sum of all parameter values
6. }
7. int main()
8. {
9.     cout << sum(10, 15) << endl; // x = 10, y = 15, z = 0, w = 0
10.    cout << sum(10, 15, 25) << endl; // x = 10, y = 15, z = 25, w = 0
11.    cout << sum(10, 15, 25, 30) << endl; // x = 10, y = 15, z = 25, w = 30
12.    return 0;
13.}
```

Output

```
25
50
80
```

Explanation

In the above program, we have called the sum function three times.

- Sum(10,15)
When this function is called, it reaches the definition of the sum. There it initializes x to 10 y to 15, and the rest values are zero by default as no value is passed. And all the values after sum give 25 as output.
- Sum(10, 15, 25)
When this function is called, x remains 10, y remains 15, the third parameter z that is passed is initialized to 25 instead of zero. And the last value remains 0. The sum of x, y, z, w, is 50 which is returned as output.
- Sum(10, 15, 25, 30)
In this function call, there are four parameter values passed into the function with x as 10, y as 15, z is 25, and w as 30. All the values are then summed up to give 80 as the output.

Note If the function is overloaded with different data types that also contain the default arguments, it may result in an ambiguous match, which results in an error.

Example

```

1. #include<iostream>
2. using namespace std;
3. int sum(int x, int y, int z=0, int w=0) // Here there are two values in the default arguments
4. { // Both z and w are initialised to zero
5.     return (x + y + z + w); // return sum of all parameter values
6. }
7. int sum(int x, int y, float z=0, float w=0) // Here sum is overloaded with two float parameter values
8. { // This results in ambiguity
9.     return (x + y + z + w);
10. }
11. int main()
12. {
13.     cout << sum(10, 15) << endl; // x = 10, y = 15, z = 0, w = 0
14.     cout << sum(10, 15, 25) << endl; // x = 10, y = 15, z = 25, w = 0
15.     cout << sum(10, 15, 25, 30) << endl; // x = 10, y = 15, z = 25, w = 30
16.     return 0;
17. }
```

Output

```

prog.cpp: In function 'int main()':
prog.cpp:15:20: error: call of overloaded 'sum(int, int)' is ambiguous
  cout << sum(10, 15) << endl; // x = 10, y = 15, z = 0, w = 0
                ^
prog.cpp:4:5: note: candidate: int sum(int, int, int, int)
  int sum(int x, int y, int z=0, int w=0) // Here there are two values in the default arguments
      ^
prog.cpp:9:5: note: candidate: int sum(int, int, float, float)
  int sum(int x, int y, float z=0, float w=0) // Here sum is overloaded with two float parameter values
      ^
```

Explanation

Here when we call the sum function with all the **parameters(x, y, z, w)** or either any one parameter value of z or w, the compiler gets confused about which function to execute. Thus, it creates an ambiguity which results in the error.

Dynamic binding in C++

The binding which can be resolved by the compiler using runtime is known as static binding. For example, all the final, static, and private methods are bound at run time. All the overloaded methods are binded using static binding.

The concept of dynamic binding removed the problems of static binding.

Dynamic binding in



Dynamic binding

The general meaning of binding is linking something to a thing. Here linking of objects is done. In a programming sense, we can describe binding as linking function definition with the function call.

So the term dynamic binding means to select a particular function to run until the runtime. Based on the type of object, the respective function will be called.

As dynamic binding provides flexibility, it avoids the problem of static binding as it happened at compile time and thus linked the function call with the function definition.

Use of dynamic binding

On that note, dynamic binding also helps us to handle different objects using a single function name. It also reduces the complexity and helps the developer to debug the code and errors.

How to implement dynamic binding?

The concept of dynamic programming is implemented with virtual functions.

Virtual functions

A function declared in the base class and overridden(redefined) in the child class is called a virtual function. When we refer derived class object using a pointer or reference to the base, we can call a virtual function for that object and execute the derived class's version of the function.

Characteristics

- Run time function resolving
- Used to achieve runtime polymorphism
- All virtual functions are declared in the base class
- Assurance of calling correct function for an object regardless of the pointer(reference) used for function call.
- A virtual function cannot be declared as static
- No virtual constructor exists but a virtual destructor can be made.
- Virtual function definition should be the same in the base as well as the derived class.
- A virtual function can be a friend of another class.
- The definition is always in the base class and overrides in the derived class

Now let us see the following problem that occurs without virtual keywords.

Example

Let us take a class A with a function **final_print()**, and class B inherits A publicly. B also has its **final_print()** function.

If we make an object of A and call **final_print()**, it will run of base class whereas, if we make an object of B and call final_print(), it will run of base only.

Code

1. `#include <iostream>`
2. `using namespace std;`
3. `class A {`
4. `public:`

```

5.   void final_print() // function that call display
6.   {
7.   display();
8.   }
9.   void display() // the display function
10.  {
11.  cout<< "Printing from the base class" <<endl;
12.  }
13. };
14. class B : public A // B inherit a publicly
15. {
16. public:
17.   void display() // B's display
18.   {
19.  cout<< "Printing from the derived class" <<endl;
20.  }
21. };
22. int main()
23. {
24.   A obj1; // Creating A's pbject
25.   obj1.final_print(); // Calling final_print
26.   B obj2; // calling b
27.   obj2.final_print();
28.   return 0;
29. }

```

Output

```

Printing from the base class
Printing from the base class

```

Now let us solve this problem using virtual function.

Code

```

1.  #include <iostream>
2.  using namespace std;
3.  class A {
4.  public:
5.   void final_print() // function that call display
6.   {
7.   display();
8.   }
9.  virtual void display() // the display function
10. {
11.  cout<< "Printing from the base class" <<endl;
12.  }
13. };
14. class B : public A // B inherit a publicly
15. {
16. public:
17.   virtual void display() // B's display
18.   {
19.  cout<< "Printing from the derived class" <<endl;
20.  }
21. };
22. int main()
23. {
24.   A obj1; // Creating A's pbject
25.   obj1.final_print(); // Calling final_print

```

```
26. B obj2; // calling b
27. obj2.final_print();
28. return 0;
29. }
```

Output

```
Printing from the base class
Printing from the derived class
```

Hence, dynamic binding links the function call with function definition with the help of virtual functions.

Dynamic memory allocation in C++

There are times where the data to be entered is allocated at the time of execution. For example, a list of employees increases as the new employees are hired in the organization and similarly reduces when a person leaves the organization. This is called managing the memory. So now, let us discuss the concept of dynamic memory allocation.



Memory allocation

Reserving or providing space to a variable is called memory allocation. For storing the data, memory allocation can be done in two ways -

- **Static allocation or compile-time allocation** - Static memory allocation means providing space for the variable. The size and data type of the variable is known, and it remains constant throughout the program.
- **Dynamic allocation or run-time allocation** - The allocation in which memory is allocated dynamically. In this type of allocation, the exact size of the variable is not known in advance. Pointers play a major role in dynamic memory allocation.

Why Dynamic memory allocation?

Dynamically we can allocate storage while the program is in a running state, but variables cannot be created "on the fly". Thus, there are two criteria for dynamic memory allocation -

- A dynamic space in the memory is needed.
- Storing the address to access the variable from the memory

Similarly, we do memory de-allocation for the variables in the memory.

In C++, memory is divided into two parts -

- **Stack** - All the variables that are declared inside any function take memory from the stack.
- **Heap** - It is unused memory in the program that is generally used for dynamic memory allocation.

Dynamic memory allocation using the new operator

To allocate the space dynamically, the operator new is used. It means creating a request for memory allocation on the free store. If memory is available, memory is initialized, and the address of that space is returned to a pointer variable.

Syntax

Pointer_variable = new data_type;

The `pointer_variable` is of pointer `data_type`. The data type can be `int`, `float`, `string`, `char`, etc.

Example

```
int *m = NULL // Initially we have a NULL pointer
```

```
m = new int // memory is requested to the variable
```

It can be directly declared by putting the following statement in a line -

```
int *m = new int
```

Initialize memory

We can also initialize memory using `new` operator.

For example

```
int *m = new int(20);
```

```
Float *d = new float(21.01);
```

Allocate a block of memory

We can also use a `new` operator to allocate a block(array) of a particular data type.

For example

```
int *arr = new int[10]
```

Here we have dynamically allocated memory for ten integers which also returns a pointer to the first element of the array. Hence, `arr[0]` is the first element and so on.

Note

- The difference between creating a normal array and allocating a block using `new` normal arrays is deallocated by the compiler. Whereas the block is created dynamically until the programmer deletes it or if the program terminates.
- If there is no space in the heap memory, the `new` request results in a failure throwing an exception(`std::bad_alloc`) until we use *nothrow* with the `new` operator. Thus, the best practice is to first check for the pointer variable.

Code

1. `int *m = new(nothrow) int;`
2. `if(!m) // check if memory is available`
3. `{`
4. `cout<< "No memory allocated";`
5. `}`

Now as we have allocated the memory dynamically. Let us learn how to delete it.

Delete operator

We delete the allocated space in C++ using the **delete** operator.

Syntax

```
delete pointer_variable_name
```

Example

```
delete m; // free m that is a variable
```

```
delete [] arr; // Release a block of memory
```

Example to demonstrate dynamic memory allocation

1. `// The program will show the use of new and delete`
2. `#include <iostream>`

```

3. using namespace std;
4. int main ()
5. {
6.     // Pointer initialization to null
7.     int* m = NULL;
8.
9.     // Request memory for the variable
10.    // using new operator
11.    m = new(nothrow) int;
12.    if (!m)
13.        cout<< "allocation of memory failed\n";
14.    else
15.    {
16.        // Store value at allocated address
17.        *m=29;
18.        cout<< "Value of m: " << *m <<endl;
19.    }
20.    // Request block of memory
21.    // using new operator
22.    float *f = new float(75.25);
23.    cout<< "Value of f: " << *f <<endl;
24.    // Request block of memory of size
25.    int size = 5;
26.    int *arr = new(nothrow) int[size];
27.    if (!arr)
28.        cout<< "allocation of memory failed\n";
29.    else
30.    {
31.        for (int i = 0; i < size; i++)
32.            arr[i] = i+1;
33.
34.        cout<< "Value store in block of memory: ";
35.        for (int i = 0; i < size; i++)
36.            cout<<arr[i] << " ";
37.    }
38.
39.    // freed the allocated memory
40.    delete m;
41.    delete f;
42.    // freed the block of allocated memory
43.    delete[] arr;
44.
45.    return 0;
46.}

```

Output

```

Value of m: 29
Value of f: 75.25
Value store in block of memory: 1 2 3 4 5

```

Fast input and output in C++

In competitive programming fast execution, input and outputs matter a lot. Sometimes we need to enter only five numbers in the array and the other time there can be 10,000. These are the cases where fast I/O comes into the picture.



Now let us discuss some useful tips beneficial to avoid the TLE -

- For taking a value as input, the general format in C++ is -
std::cin>>x; // x is the value to be input
The cin works better for a range of inputs but it is convenient to use -
scanf("%d",&x); // x is the value to be input
Similarly, for printing on the console we use -
Std::cout<<x ; // x is the value to be printed
The cout also works fine for limited numbers. The best practice is using-
printf("%d", x); // x is the value to be printed
- To gain the same speed of scanf/printf with cin/cout. Add the below lines in the main function -
ios_base::sync_with_stdio(false) - It toggles all the synchronization of all C++ with their respective C streams when called before cin/cout in a program. We make this function false(which is true earlier) to avoid any synchronization.
cin. tie(NULL) - The function helps to flush all the std::cout before any input is taken. It is beneficial in interactive console programs in which the console is updated regularly but also slows down the program for larger values. NULL refers to returning a null pointer.
- Include a header file that contains all the others libraries. It is a standard header for the GNU C++ library. Including the below header file saves time and the effort to add a particular library for a data structure. For example, for declaring a map we need a header file. This extra effort is reduced which is also time saving.
#include

This is how a common C++ program template looks on applying the tips:

- #include <bits/stdc++.h> // GNU header**
- using namespace std;**
- int main()**
- {**
- ios_base::sync_with_stdio(false);**
- cin.tie(NULL); // flushes cout**
- return 0;**
- }**

- It is advised to use "\n" for coming to the new line instead of endl.
"\n" - Enters a new line
endl - Enters a new line and flushes the stream
Therefore,
cout<<endl = cout<< "\n" << flush

Differences between "\n" and endl

endl	\n
Manipulator	Character
Does not occupy memory	Occupy memory of 1 byte
endl is a keyword. So it won't give meaning when stored in a string	\n can be stored in a string
Supported only by C++	Supported by both

Hierarchical inheritance in C++

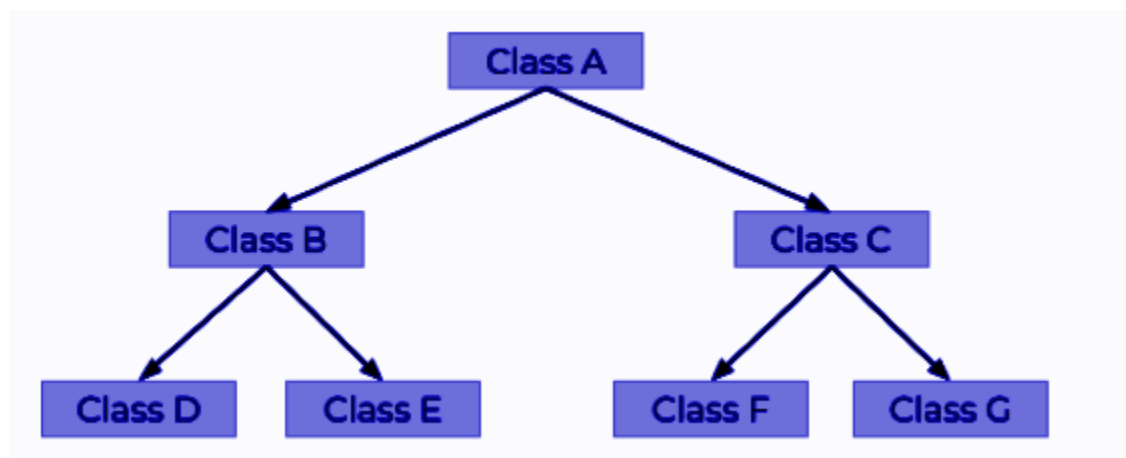
The concept of inheritance is very similar to the real world. Just like a son inherits the properties (characteristics and behavior) of his father and father himself inherits the properties of the son's grandfather. In programming norms, inheritance occurs when one class inherits the properties of another class(base).

Now let us understand the concept of hierarchical inheritance.



Definition

As the name defines, it is the hierarchy of classes. There is a single base class and multiple derived classes. Furthermore, the derived classes are also inherited by some other classes. Thus a tree-like structure is formed of hierarchy.



Here class A is the base class. Class B and Class C are the derived classes of A.

Class D and Class E are derived classes of B. Class F and Class G are derived classes of C. Thus forming the structure of hierarchical inheritance.

Where hierarchical inheritance does is used?

It is used in the following cases where hierarchy is to be maintained. For instance, the database of an organization is stored in the hierarchical format. There are different sections of an organization such as IT, computer science, Civil, Mechanical, etc. Each organization has the same attributes such as student name, roll number, year, etc. which comes under a class Student. Hence all the sections inherit the student properties and thus following the format of hierarchical inheritance.

Syntax

1. Class Parent
2. {
3. statement(s);
4. };
5. Class Derived1: **public** Parent
6. {
7. statement(s);
8. };
9. Class Derived2: **public** Parent
10. {
11. statement(s);
12. };
13. **class** newderived1: **public** Derived1
14. {

```
15. statement(s);
16. };
17. class newderived2: public Derived2
18. {
19. statement(s);
20. };
```

Class Parent is the base class and Derived1 and Derived2 are the class that inherit Parent class. Further, newderived1 is the class that inherits Derived1, and newderived2 is the class that inherits the Derived2 class. There can be any number of base classes that are inherited by n number of derived classes.

Code example

```
1. #include <iostream>
2. using namespace std;
3. class A // Base class
4. {
5. public:
6. int x, y; // data members
7. void getdata() // to input x and y
8. {
9. cout << "Enter value of x and y:\n";
10. cin >> x >> y;
11. }
12. };
13. class B : public A // B is derived from class base
14. {
15. public:
16. void product()
17. {
18. cout << "\nProduct= " << x * y << endl; // Perform product
19. }
20. };
21. class C : public A // C is also derived from class base
22. {
23. public:
24. void sum()
25. {
26. cout << "\nSum= " << x + y; // Perform sum
27. }
28. };
29. int main()
30. {
31. B obj1; // object of derived class B
32. C obj2; // object of derived class C
33. obj1.getdata(); // input x and y
34. obj1.product();
35. obj2.getdata();
36. obj2.sum();
37. return 0;
38. }
```

Output

```
/tmp/dak8cBbqw5.o
Enter value of x and y:
2 3

Product= 6
Enter value of x and y:
5 6

Sum= 11|
```

Explanation

We have class A as our base class which has two data members x and y. It also inputs the values of the data members using the function `getdata()`. Class B inherits class A and performs the products using the inherited data members x and y.

Class C also calls the `getdata()` of base and performs the product of x and y using the inherited data members.

Hybrid inheritance in C++

Inheritance is defined as the process in which one class inherits the property of another class. The class whose property is inherited is called as Base class or the parent of that class. The class that inherits the base class's properties (parent class) is the derived class.

For example, A son inherits the properties of his father. This article will give you a brief introduction to hybrid inheritance and its examples.



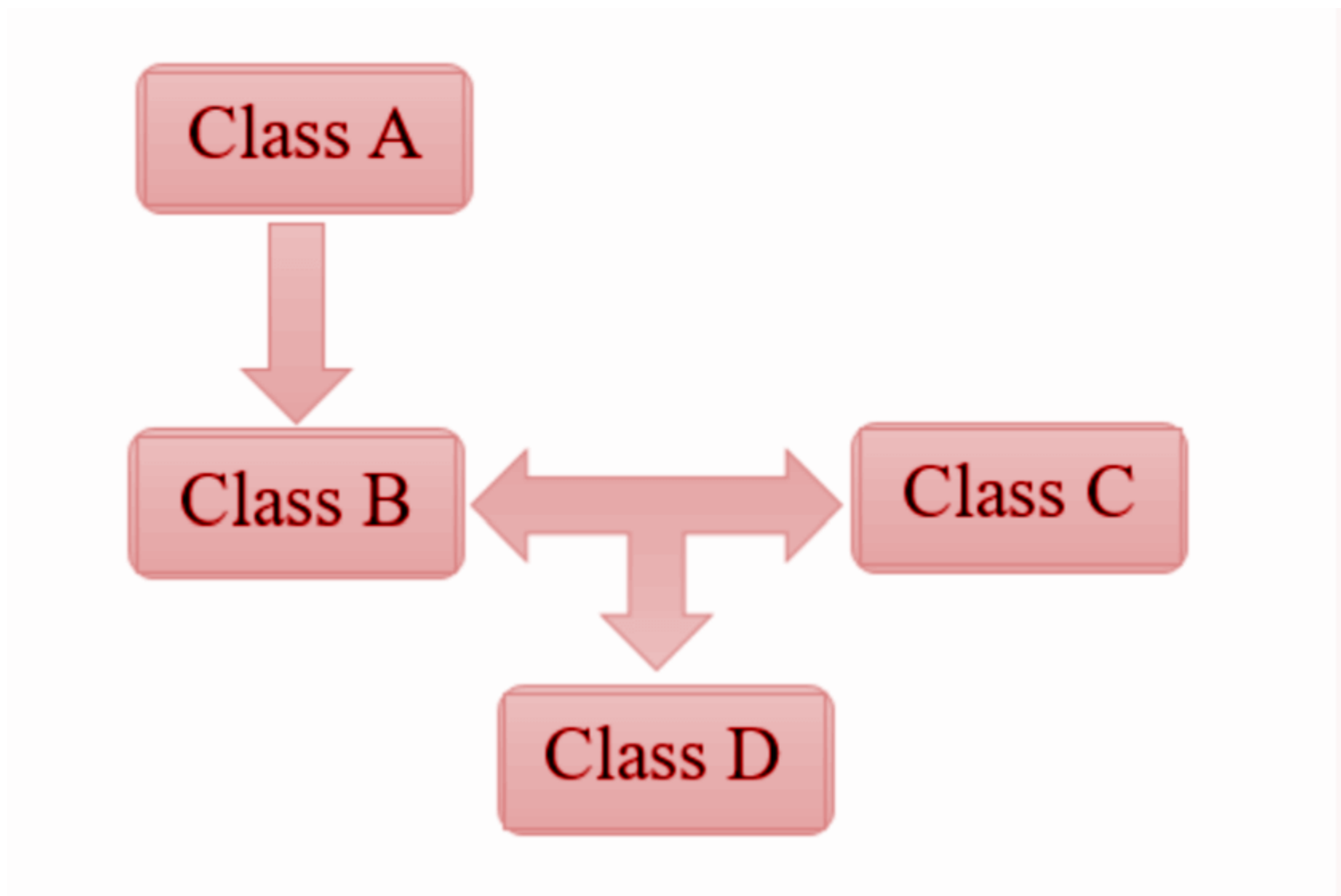
Definition

Combining various types of inheritance like multiple, simple, and hierarchical inheritance is known as hybrid inheritance.

In simple inheritance, one class is derived from a single class which is its base. In multiple inheritances, a class is derived from two classes, where one of the parents is also a derived class. In hierarchical inheritance, more than one derived class is created from a single base class.

In hybrid inheritance, there is a combination of one or more inheritance types. For instance, the combination of single and hierarchical inheritance. Therefore, hybrid inheritance is also known as multipath inheritance.

Example



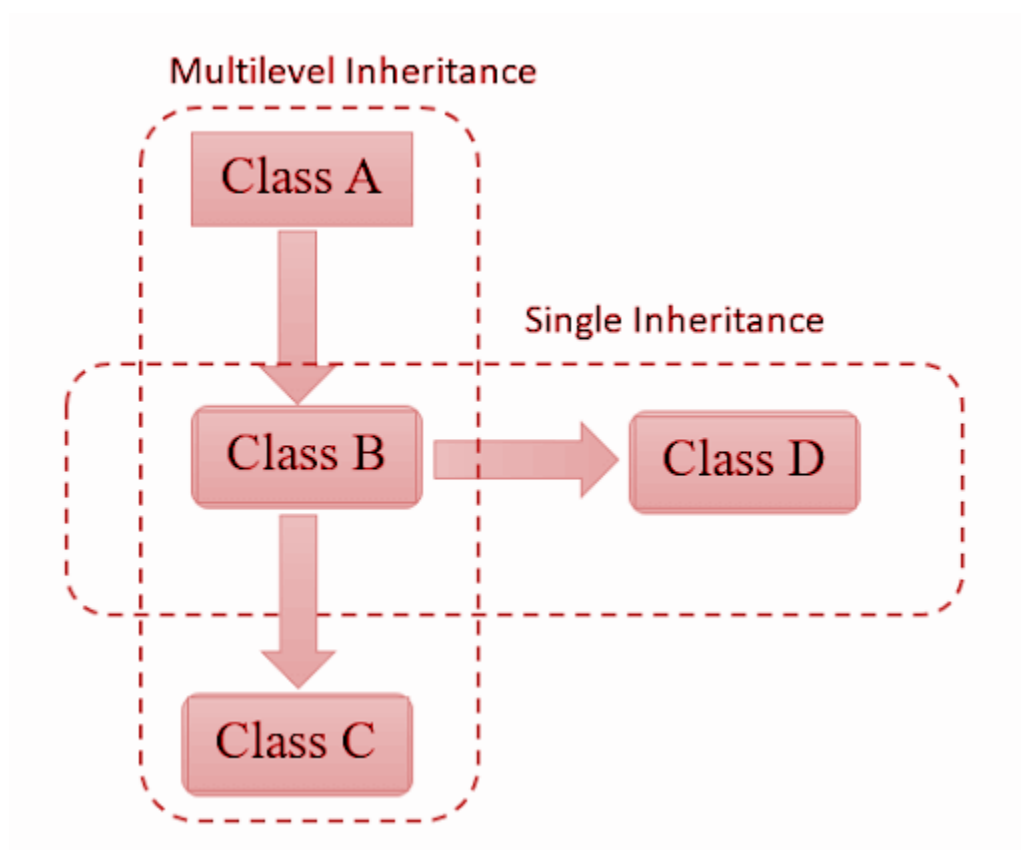
The diagram shows the hybrid inheritance that is a combination of single inheritance and multiple inheritance.

Single inheritance - Class B inherits class A. Thus an example of single inheritance.

Multiple inheritance - Class D is inherited from multiple classes(B and C shown above D). Thus an example of multiple inheritance.

Syntax code of the above example

```
1. Class A
2. {
3. statement(s)
4. };
5. Class B: public A
6. {
7. statement(s);
8. };
9. Class C
10. {
11. statement(s);
12. };
13. Class D: public B, public C
14. {
15. statement(s);
16. };
```



This example shows the combination of multilevel and single inheritance.

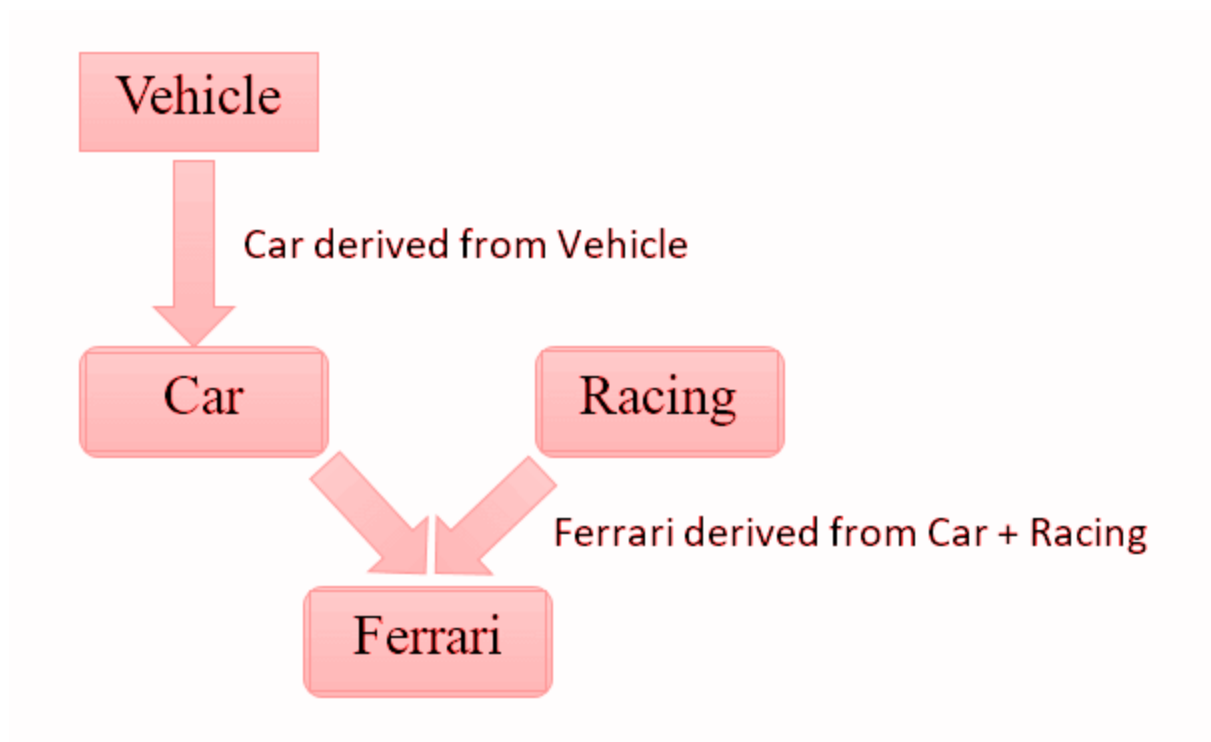
Multilevel inheritance - As seen from the above diagram, Class B inherits class A and class C inherits class B. Thus, it is an example of multilevel inheritance.

Single inheritance - From the above diagram, Class D inherits class B. Thus, it is an example of single inheritance.

Syntax code for the above example

1. Class A
2. {
3. statement(s);
4. };
5. Class B: **public** A
6. {
7. statement(s);
8. };
9. Class C: **public** B
10. {
11. statement(s);
12. };
13. Class D: **public** B
14. {
15. statement(s);
16. };

A real-life example of hybrid inheritance



In a real-world scenario, we all drive a Car. So **Car** is a class that comes under **vehicle** class. Thus an instance of single inheritance.

If we talk about the **Ferrari**, that is a combination of the racing car and a normal car. So **class Ferrari** is derived from the class **Car** and **Class Racing**.

Hence, the above example is a single and multiple inheritance. It is a perfect example of hybrid inheritance (single + multiple).

Code

```
1. #include <iostream>
2. using namespace std;
3. class vehicle
4. {
5. public:
6. vehicle()
7. {
8. cout<< "This is a vehicle\n";
9. }
10.};
11. class Car: public vehicle
12. {
13. public:
14. Car()
15. {
16. cout<< "This is a car\n";
17. }
18.};
19. class Racing
20. {
21. public:
22. Racing()
23. {
24. cout<< "This is for Racing\n";
25. }
26.};
27. class Ferrari: public Car, public Racing
28. {
29. public:
30. Ferrari()
31. {
32. cout<< "Ferrari is a Racing Car\n";
33. }
```

```

34.
35. };
36. int main() {
37.     Ferrari f;
38.     return 0;
39. }

```

Output

```

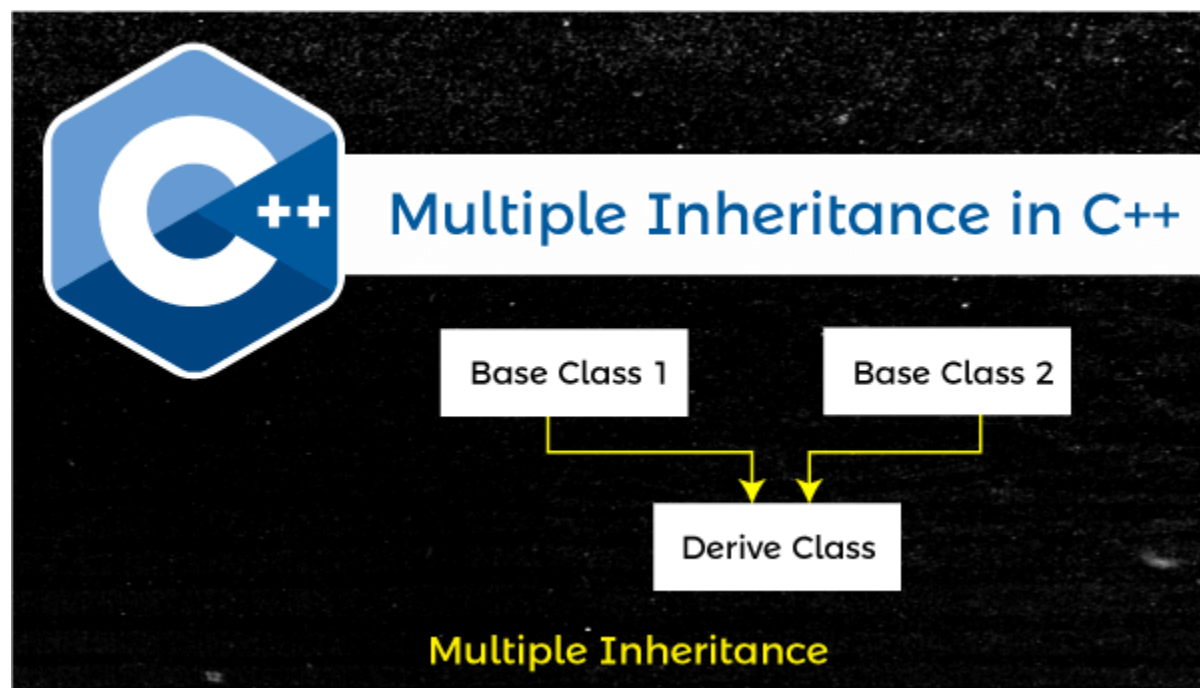
This is a vehicle
This is a car
This is for Racing
Ferrari is a Racing Car

```

Multiple Inheritance in C++

This section will discuss the Multiple Inheritances in the C++ programming language. When we acquire the features and functionalities of one class to another class, the process is called **Inheritance**. In this way, we can reuse, extend or modify all the attributes and behaviour of the parent class using the derived class object. It is the most important feature of object-oriented programming that reduces the length of the program.

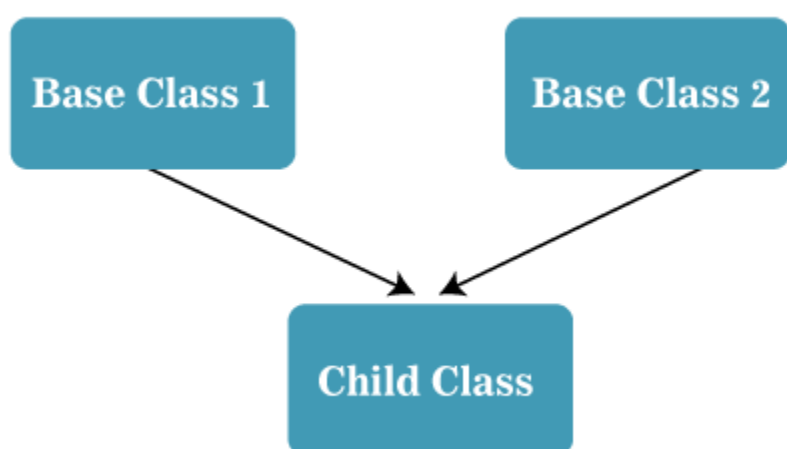
A class that inherits all member functions and functionality from another or parent class is called the **derived class**. And the class from which derive class acquires some features is called the **base or parent class**.



Multiple Inheritance is the concept of the Inheritance in C++ that allows a child class to inherit properties or behaviour from multiple **base** classes. Therefore, we can say it is the process that enables a derived class to acquire member functions, properties, characteristics from more than one base class.

Diagram of the Multiple Inheritance

Following is the diagram of the Multiple Inheritances in the [C++ programming language](#).



In the above diagram, there are two-parent classes: **Base Class 1** and **Base Class 2**, whereas there is only one **Child Class**. The Child Class acquires all features from both Base class 1 and Base class 2. Therefore, we termed the type of Inheritance as Multiple Inheritance.

Syntax of the Multiple Inheritance

1. **class** A

```

2. {
3. // code of class A
4. }
5. class B
6. {
7. // code of class B
8. }
9. class C: public A, public B (access modifier class_name)
10. {
11. // code of the derived class
12. }

```

In the above syntax, class A and class B are two base classes, and class C is the child class that inherits some features of the parent classes.

Let's write the various program of Multiple Inheritance to inherit the member functions and functionality from more than one base class using the derived class.

Example 1: Program to use the Multiple Inheritance

Program1.cpp

```

1. #include <iostream>
2. using namespace std;
3.
4. // create a base class1
5. class Base_class
6. {
7. // access specifier
8. public:
9. // It is a member function
10. void display()
11. {
12. cout << " It is the first function of the Base class " << endl;
13. }
14. };
15.
16. // create a base class2
17. class Base_class2
18. {
19. // access specifier
20. public:
21. // It is a member function
22. void display2()
23. {
24. cout << " It is the second function of the Base class " << endl;
25. }
26. };
27.
28. /* create a child_class to inherit features of Base_class and Base_class2 with access specifier. */
29. class child_class: public Base_class, public Base_class2
30. {
31.
32. // access specifier
33. public:
34. void display3() // It is a member function of derive class
35. {
36. cout << " It is the function of the derived class " << endl;
37. }

```



```

38.
39. };
40.
41. int main ()
42. {
43.     // create an object for derived class
44.     child_class ch;
45.     ch.display(); // call member function of Base_class1
46.     ch.display2(); // call member function of Base_class2
47.     ch.display3(); // call member function of child_class
48. }

```

Output

```

It is the first function of the Base class
It is the second function of the Base class
It is the function of the derived class

```

In the above program, we created two base classes and one child class. The **child_class** invoke member function display() and display2() from both parent classes **Base_class** and **Base_class2** with the help of child class's object ch.

Example 2: Use Multiple Inheritance to perform the arithmetic operation

Let's create a derived class to inherit the member functions from multiple base classes in C++ programming.

Program2.cpp

```

1. #include <iostream>
2. using namespace std;
3.
4. // create add class
5. class add
6. {
7.     public:
8.         int x = 20;
9.         int y = 30;
10.        void sum()
11.        {
12.            cout << " The sum of " << x << " and " << y << " is " <<x+y << endl;
13.        }
14. };
15.
16. // create Mul class
17. class Mul
18. {
19.     public:
20.         int a = 20;
21.         int b = 30;
22.         void mul()
23.         {
24.             cout << " The Multiplication of " << a << " and " << b << " is " <<a*b << endl;
25.         }
26. };
27.
28. // create Subclass
29. class Sub
30. {
31.     public:
32.         int a = 50;
33.         int b = 30;
34.         void sub()

```

```

35.     {
36.         cout << " The Subtraction of " << a << " and " << b << " is " <<a-b << endl;
37.     }
38. };
39.
40. // create Div class
41. class Div
42. {
43.     // access specifier
44.     public:
45.         int a = 150;
46.         int b = 30;
47.         void div()
48.         {
49.             cout << " The Division of " << a << " and " << b << " is " <<a/b << endl;
50.         }
51. };
52.
53. // create a derived class to derive the member functions of all base classes
54. class derived: public add, public Div, public Sub, public Mul
55. {
56.     // access specifier
57.     public:
58.         int p = 12;
59.         int q = 5;
60.         void mod()
61.         {
62.             cout << "The Modulus of " << p << " and " <<q << " is " << p % q << endl;
63.         }
64. };
65.
66. int main ()
67. {
68.     // create an object of the derived class
69.     derived dr;
70.     dr.mod(); // call derive class member function
71.     // call all member function of class add, Div, Sub and Mul
72.     dr.sum();
73.     dr.mul();
74.     dr.div();
75.     dr.sub();
76. }

```

Output

```

The Modulus of 12 and 5 is 2
The sum of 20 and 30 is 50
The Multiplication of 20 and 30 is 600
The Division of 150 and 30 is 5
The Subtraction of 50 and 30 is 20

```

Example 3: Get the average marks of six subjects using the Multiple Inheritance

Let's create another program to print the average marks of six subjects using the multiple Inheritance in the C++ programming language.

Program3.cpp

```

1. #include <iostream>
2. using namespace std;
3.
4. // create base class1

```

```

5. class student_detail
6. {
7.     // access specifier
8.     protected:
9.         int rno, sum = 0, i, marks[5];
10.
11.    // access specifier
12.    public:
13.        void detail()
14.        {
15.
16.
17.            cout << " Enter the Roll No: " << endl;
18.            cin >> rno;
19.
20.            cout << " Enter the marks of five subjects " << endl;
21.            // use for loop
22.            for (i = 0; i < 5; i++)
23.            {
24.                cin >> marks[i];
25.            }
26.
27.
28.            for ( i = 0; i < 5; i++)
29.            {
30.                // store the sum of five subject
31.                sum = sum + marks[i];
32.            }
33.        }
34.
35.};
36.
37.// create base class2
38.class sports_mark
39.{
40.    protected:
41.        int s_mark;
42.
43.    public:
44.
45.        void get_mark()
46.        {
47.            cout << "\n Enter the sports mark: ";
48.            cin >> s_mark;
49.        }
50.};
51.
52./* create a result as the child class to inherit functions of the parent class: student_detail and sports_mark.*/
53.class result: public student_detail, public sports_mark
54. {
55.     int tot, avg;
56.     public:
57.
58.     // create member function of child class
59.     void disp ()
60.     {
61.         tot = sum + s_mark;

```

```

62.         avg = tot / 6; // total marks of six subject / 6
63.         cout << " \n \n \t Roll No: " << rno << " \n \t Total: " << tot << endl;
64.         cout << " \n \t Average Marks: " << avg;
65.     }
66. };
67.
68. int main ()
69. {
70.     result obj; // create an object of the derived class
71.
72.     // call the member function of the base class
73.     obj.detail();
74.     obj.get_mark();
75.     obj.disp();
76. }

```

Output

```

Enter the Roll No:
25
Enter the marks of five subjects
90
85
98
80
75

Enter the sports mark: 99

Roll No: 25
Total: 527

Average Marks: 87

```

Ambiguity Problem in Multiple Inheritance

In Multiple Inheritance, when a single class is derived from two or more base or parent classes. So, it might be possible that both the parent class have the same-named member functions, and it shows ambiguity when the child class object invokes one of the same-named member functions. Hence, we can say, the C++ compiler is confused in selecting the member function of a class for the execution of a program.

Program to demonstrate the Ambiguity Problem in Multiple Inheritance

Let's write a simple to invoke the same member function of the parent class using derived class in C++ programming.

Program4.cpp

```

1. #include <iostream>
2. #include <conio.h>
3.
4. using namespace std;
5.
6. // create class A
7. class A
8. {
9.     public:
10.    void show()
11.    {
12.        cout << " It is the member function of class A " << endl;
13.    }
14. };
15.
16. // create class B
17. class B
18. {
19.     public:

```

```

20. void show()
21. {
22.     cout << " It is the member function of class B " << endl;
23. }
24. };
25.
26.
27. // create a child class to inherit the member function of class A and class B
28. class child: public A, public B
29. {
30.     public:
31.         void disp()
32.         {
33.             cout << " It is the member function of the child class " << endl;
34.         }
35. };
36.
37. int main ()
38. {
39.     // create an object of the child class to access the member function
40.     child ch;
41.     ch.show(); // It causes ambiguity
42. ch.disp();
43.     return 0;
44. }

```

When the above program is compiled, it throws the show() member function is ambiguous. Because of both the base class A and B, defining the same member function show(), and when the derived class's object call the shows() function, it shows ambiguity in multiple inheritances.

Therefore, we need to resolve the ambiguous problem in multiple Inheritance. The ambiguity problem can be resolved by defining the **class name** and **scope resolution (::) operator** to specify the class from which the member function is invoked in the child class.

Syntax of the Ambiguity Resolution

1. Derived_obj_name.parent_class_name :: same_named_memberFunction ([parameter]);

For example,

1. ch.A:: show(); // class_name and scope resolution operator with member function
2. ch.B::show();

After making some changes, now we again execute the above program that returns the given below Output.

1. It is the member function of the child **class**
2. It is the member function of **class** A
3. It is the member function of **class** B

C++ Bitwise XOR Operator

- Bitwise XOR operator is also known as **Exclusive OR**
- It is denoted by using the '^'
- As the name depicts, it works on the **bit level** of the operands.
- Bitwise XOR operator has come under the category of Bitwise operators.
- In the bitwise exclusive OR operator (XOR), two operands are required, and these two operands are separated by the XOR symbol, i.e., '^'.
- To determine the output or result that comes out after applying the XOR operator on two operands, we need to follow the Logical truth table of the XOR operator.

- XOR Truth Table is the mathematical table constructed using the proper logic of the XOR operator.
- The logic used behind the XOR operator is; whenever XOR operation is applied on the two **different** bits of two operands, then the result will always produce '**1**', and if the XOR operation is applied on the two **same** bits of two operands then the result produces output '**0**'.

Truth Table of Exclusive OR (XOR) operator

Let there are two operands; the First one is A and the second one is B, the total combinations of input formed by these two operands will be 4. By using the following XOR truth table, we will determine the corresponding output. The result will be captured in C, here **C = A ^ B**.

In this truth table, we are taking input in the form of bits, i.e., 0 and 1, and the output will also be generated in the form of bits, i.e., 0 and 1.

A	B	C = A ^ B
0	0	0
0	1	1
1	0	1
1	1	0

Here, in the above XOR Truth table, we observe that when the values of operands A and B are different, i.e., (0, 1), (1, 0), the result that comes out will always be 1. And when the values of operands A and B are the same, i.e., (0, 0), (1, 1), the result that comes out will always be 0.

Similarly, in this way, we can draw the truth table for **Boolean** values -

Let there are two operands; the First one is **A** and the second one is **B**. The total combinations of input formed by these two operands will be 4. By using the following XOR truth table, we will determine the corresponding output. The result will be captured in C, here $C = A \wedge B$.

In this truth table, we are taking input in the form of Truth values, i.e., True (T) and False (F). The output will also be generated in the form of True values, i.e., T and F.

A	B	C = A ^ B
F	F	F
F	T	T
T	F	T
T	T	F

Here, in the above XOR Truth table, we observe that, when the values of operands A and B are different, i.e., (F, T), (T, F), the result comes out will always be T. And when the values of operands A and B are same, i.e., (F, F), (T, T), the result comes out will always be F.

From the tables above, we observe that **T (True) is denoted by one and F (False) is denoted by 0**.

Steps to solve any given problem -

1. The operands given in the problem will always be in the decimal value.
2. Firstly, we need to convert the values of operands into **binary**
3. After converting the values of operands in binary numbers, put both operands one over each other.
4. Remember that before applying exclusive OR (XOR) operation on them, kindly check the **number of digits** in them.
5. If the count of digits does not match, the extra 0's at the left end of the small operand balances the counts of digits.

- 6. Finally, with the help of the above truth table, apply the XOR operation on the operands one by one, taking one bit at a time for applying the XOR operation.
- 7. At last, the result is produced in the form of output.
- 8. The output is produced will be in binary form, now convert the binary form into decimal form and note down the result value.

Execution of Bitwise Exclusive OR (XOR) operation in C++

Let us understand in more detail about the execution of the XOR operation in C++ with the help of examples -

Example 1: Find the exclusive OR of integer values; 10 and 14. Also, explain it and write the code of execution in C++.

Solution: Let us consider two variables, ' a ' and ' b ', to store the corresponding two operands given in the above question, i.e., 10 and 14.

Here, a = 10 and b = 14.

We will follow the below steps to find out the exclusive OR of the given two operands.

- 1. We know that 10 and 14 are in decimal form, and for applying bitwise XOR operation, it is necessary to convert it into binary form.
- 2. Binary form ' a ', i.e., 10 is ' 1010 ' and Binary form of ' b ', i.e., 14 is ' 1110 '.
- 3. Here we observe that the count of binary digits present in a is four and the count of binary digits present in b is also 4; hence the number of binary digits present in both the variables are the same and already balanced, we do not need to add more number of 0's to balance it.
- 4. Now, putting the binary digits present in 'b' down to the binary digits present in 'a'.
- 5. Finally, applying the XOR operation one by one on the corresponding bits matches and note down the output.
- 6. The output generated at last will be in binary form, as the above question given in the decimal form, so we need to convert the result in decimal form.

Explanation:

a = 10 (In Decimal form)

b = 14 (In Decimal form)

Now, for an XOR b, we need to convert a and b in binary form -

a = 1010 (In Binary form)

b = 1110 (In Binary form)

Now, applying XOR operation on a and b -

a = 1010

b = 1110

a ^ b = 0100 (In Binary form)

The result of a ^ b is 0100, which is in binary form.

Now converting the result in decimal form, which is 4.

10 ^ 14 = 4

NOTE: By using the above XOR truth table, the output of corresponding bits are generated.

We will now apply the bitwise XOR operation on 10 and 14 in C++ language and get the result, i.e., 4.

C++ code for above example:

- 1. `//***** C++ Code *****`
- 2. `#include<iostream>`

```

3.  using namespace std;
4.  int main ()
5.  {
6.      int a, b, c;           // Initializing integer variables to store data values
7.      cout << "Enter values of a and b -> " << endl;
8.      cout << "a: ";
9.      cin >> a;              // taking a as input from user
10.     cout << "b: ";
11.     cin >> b;              // taking b as input from user
12.     c = a ^ b;             // storing XOR result of a and b in c
13.     cout << "Applying XOR operation on a and b: " << endl;
14.     cout << "a ^ b = " << c << endl;      // Printing the output
15.
16. }

```

Output

```

Enter values of a and b ->
a: 10
b: 14
Applying XOR operation on a and b:
a ^ b = 4

-----
Process exited after 4.469 seconds with return value 0
Press any key to continue . . .

```

Example 2: Find the exclusive OR of integer values; 3 and 15. Also, explain it and write the code of execution in C++.

Solution: Let us consider two variables, 'a' and 'b', to store the corresponding two operands given in the above question, i.e., 3 and 15.

Here, a = 3 and b = 15.

We will follow the below steps to find out the exclusive OR of the given two operands.

1. We know that 3 and 15 are in decimal form, and for applying bitwise XOR operation, it is necessary to convert it into binary form.
2. Binary form 'a', i.e., 3 is '11' and Binary form of 'b', i.e., 15 is '1111'.
3. Here we will observe that the count of binary digits present in a is two and the count of binary digits present in b is four; hence the number of binary digits present in both the variables are not the same. Thus, unbalanced, we do need to add more number of 0's on the left side of the lower binary number, i.e., a, which is '11', to balance it.
4. After balancing, the value of a is '0011', and b is '1111'.
5. Now, putting the binary digits present in 'b' down to the binary digits present in 'a'.
6. Finally, applying the XOR operation one by one on the corresponding bits matches and note down the output.
7. The output generated at last will be in binary form, as the above question given in the decimal form, so we need to convert the result in decimal form.

Explanation:

a = 3 (In Decimal form)

b = 15 (In Decimal form)

Now, for an XOR b, we need to convert a and b in binary form -

a = 0011 (In Binary form)

b = 1111 (In Binary form)

Now, applying XOR operation on a and b -

a = 0011

b = 1111

a ^ b = 1100 (In Binary form)

The result of a ^ b is 1100, which is in binary form.

Now converting the result in decimal form, which is 12.

3 ^ 15 = 12

NOTE: By using the above XOR truth table, the output of corresponding bits are generated.

We will now apply the bitwise XOR operation on 3 and 15 in C++ language and get the result, i.e., 12.

C++ code for above example:

```
1.  1. //***** C++ Code *****
2.  2. #include<iostream>
3.  3. using namespace std;
4.  4. int main ()
5.  5. {
6.    6. int a, b, c;           // Initializing integer variables to store data values
7.    7. cout << "Enter values of a and b -> " << endl ;
8.    8. cout << "a: ";
9.    9. cin >> a;           // taking a as input from user
10.   10. cout << "b: ";
11.   11. cin >> b;           // taking b as input from user
12.   12. c = a ^ b;           // storing XOR result of a and b in c
13.   13. cout << "Applying XOR operation on a and b: " << endl ;
14.   14. cout << "a ^ b = " << c << endl ;           // Printing the output
15.   15.
16. 16. }
```

Output

```
Enter values of a and b ->
a: 3
b: 15
Applying XOR operation on a and b:
a ^ b = 12

-----
Process exited after 6.107 seconds with return value 0
Press any key to continue . . .
```

Different Ways to Compare Strings in C++

This section will discuss the different ways to compare the given strings in the C++ programming language. The comparison of the string determines whether the first string is equal to another string or not. Example: HELLO and Hello are two different strings.



There are different ways to compare the strings in the C++ programming language, as follows:

1. Using strcmp() function
2. Using compare() function
3. Using Relational Operator
4. Using For loop and If statement
5. Using user-defined function

strcmp() function

The strcmp() is a pre-defined library function of the **string.h** header file. The strcmp() function compares two strings on a lexicographical basis. This means that the strcmp() function starts comparing the first string with the second string, character by character until all characters in both strings are the same or a NULL character is encountered.

Syntax

1. **int** strcmp (**const char** *leftstr, **const char** *rightstr);

Parameters:

leftstr: It defines the characters of the left string.

rightstr: It defines the characters of the right string.

Returns:

The leftstr string compares each character with the second string from the left side till the end of both strings. And, if both the strings are equal, the strcmp() function returns strings are equal. Else, the strings are not equal.

Let's create a program to compare strings using the strcmp() function in C++.

Program1.cpp

```
1. #include <iostream>
2. using namespace std;
3. #include <string.h>
4.
5. int main ()
6. {
7.     // declare strings
8.     const char *str1 = " Welcome to JavaTpoint";
9.     const char *str2 = " Welcome to JavaTpoint";
10.
11.    const char *str3 = " JavaTpoint";
12.    const char *str4 = " Javatpoint";
13.
14.    cout << " String 1: " << str1 << endl;
15.    cout << " String 2: " << str2 << endl;
16.
```

```

17. // use strcmp() function to validate the strings are equal
18. if (strcmp (str1, str2) == 0)
19. {
20.     cout << " \n Both strings are equal. " << endl;
21. }
22. else
23. {
24.
25.     cout << " The strings are not equal. " << endl;
26. }
27.
28. cout << " \n String 3: " << str3 << endl;
29. cout << " String 4: " << str4 << endl;
30.
31. // use strcmp() function to validate the strings are equal
32. if (strcmp (str3, str4) == 0)
33. {
34.     cout << " \n Both strings are equal. " << endl;
35. }
36. else
37.     cout << " \n The strings are not equal. ";
38.
39. return 0;
40. }

```

Output

```

String 1:  Welcome to JavaTpoint
String 2:  Welcome to JavaTpoint

Both strings are equal.

String 3:  JavaTpoint
String 4:  Javatpoint

The strings are not equal.

```

compare() function

The compare() function is a pre-defined library function of the C++ language. The compare() function compares two given strings and returns the following results based on the matching cases:

1. If both the strings are the same, the function returns 0.
2. If the character value of the first string is smaller than the second string, the function returns < 0.
3. If the second string is greater than the first string, the function returns greater than 0 or >0.

Syntax

1. **int** compare (**const** string &str) **const**;

Let's create a simple program to compare two strings using the compare() function in C++.

Program2.cpp

```

1. #include <iostream>
2. using namespace std;
3. int main ()
4. {
5.     string str1, str2; // declare string variable
6.
7.     cout << " Enter the string 1: ";
8.     cin >> str1;
9.
10.    cout << " Enter the string 2: ";

```

```

11.  cin >> str2;
12.
13.  // use compare() function to compare the second string with first string
14.
15.  int i = str1.compare(str2);
16.
17.  if ( i < 0)
18.  {
19.      cout << str1 << " is smaller than " << str2 << " string" << endl;
20.  }
21.
22.  else if ( i > 0)
23.  {
24.      cout << str2 << " is greater than " << str1 << " string." << endl;
25.  }
26.  else // i == 0;
27.  {
28.      cout << " Both strings are equal.";
29.  }
30. return 0;
31.}

```

Output

```

1st Run:
Enter the string 1: Program
Enter the string 2: program
Program is smaller than program string

2nd Run:
Enter the string 1: APPLE
Enter the string 2: APPLE
Both strings are equal.

```

Relational Operator

It is the operator used to compare two strings or numerical values in C++. C++ has different types of relational operators such as '==', '!=', '>', '<' operator. But here, we use only two operators such as '==' equal to and '!=' not equal to a relational operator to compare the string easily.

Syntax

1. String1 == string2 // here, we use double equal to operator
2. Or
3. String1 != string2 // here, we use not equal to operator

Compare two strings using the Equal to (==) operator in C++

Equal To (==) operator: It is used to check the equality of the first string with the second string.

Let's create a program to compare strings using the double equal to (==) operator in C++.

Program3.cpp

```

1.  #include <iostream>
2.  using namespace std;
3.
4.  int main ()
5.  {
6.      // declare string variables
7.      string str1;
8.      string str2;
9.
10.     cout << " Enter the String 1: " << endl;
11.     cin >> str1;

```

```

12. cout << " Enter the String 2: " << endl;
13. cin >> str2;
14.
15. // use '==' equal to operator to check the equality of the string
16. if ( str1 == str2)
17. {
18.     cout << " String is equal." << endl;
19. }
20. else
21. {
22.     cout << " String is not equal." << endl;
23. }
24. return 0;
25.}

```

Output

```

Enter the String 1:
JavaTpoint
Enter the String 2:
javatpoint
String is not equal.

```

2nd Execution:

```

Enter the String 1:
Program
Enter the String 2:
Program
String is equal.

```

Compare two strings using the Not Equal To (!=) Relational Operator

Let's create a program to compare whether the strings are equal or not using the Not Equal To (!=) operator in C++.

Program4.cpp

```

1. #include <iostream>
2. using namespace std;
3.
4. int main ()
5. {
6.     // declare string variables
7.     string str1;
8.     string str2;
9.
10.    cout << " Enter the String 1: " << endl;
11.    cin >> str1;
12.    cout << " Enter the String 2: " << endl;
13.    cin >> str2;
14.
15.    // use '!=' not equal to operator to check the equality of the string
16.    if ( str1 != str2)
17.    {
18.        cout << " String is not equal." << endl;
19.    }
20.    else
21.    {
22.        cout << " String is equal." << endl;
23.    }
24.    return 0;
25.}

```

Output

```
Enter the String 1:
JAVATpoint
Enter the String 2:
JavaTPOINT
String is not equal.
```

2nd Run:

```
Enter the String 1:
HELLO
Enter the String 2:
HELLO
String is equal.
```

Compare two strings using for loop and if statement in C++

Program5.cpp

```
1. #include <iostream>
2. using namespace std;
3. int main ()
4. {
5.     char s1[50], s2[50]; // declare character array
6.     int i, disp;
7.
8.     cout << " Enter the String 1: " << endl;
9.     cin >> s1;
10.
11.    cout << " Enter the String 2: " << endl;
12.    cin >> s2;
13.
14.    for (i = 0; s1[i] == s2[i] && s1[i] == '\0'; i++);
15.
16.    if (s1[i] < s2[i])
17.    {
18.        cout << " String 1 is less than String 2";
19.    }
20.    else if (s1[i] > s2[i])
21.    {
22.        cout << " String 2 is less than String 1";
23.    }
24.    else
25.    {
26.        cout << " String 1 is equal to String 2";
27.    }
28.    return 0;
29. }
```

Output

```
Enter the String 1:
WELCOME
Enter the String 2:
WELCOME
String 1 is equal to String 2
```

Compare two strings using the User-defined function in C++

Let's create a simple program to compare the first string with another string using the user-defined function in C++.

Program6.cpp

```
1. #include <iostream>
2. using namespace std;
3.
4. void RelationalCompare ( string str1, string str2)
5. {
6.     // use relational not equal operator
```

```
7.  if ( str1 != str2)
8.  {
9.      cout << str1 << " is not equal to " << str2 << " string." << endl;
10.  if (str1 > str2)
11.  {
12.      cout << str1 << " is greater than " << str2 << " string." << endl;
13.  }
14.  else
15.  {
16.      cout << str2 << " is greater than " << str1 << " string." << endl;
17.  }
18. }
19.  else
20.      cout << str1 << " is equal to " << str2 << " string." << endl;
21.}
22.
23.int main ()
24.{
25.    string str1 ( "JavaT");
26.    string str2 ( "Tpoint");
27.
28.    // call function
29.    RelationalCompare (str1, str2);
30.
31.    string str3 ("JavaTpoint");
32.    string str4 ("JavaTpoint");
33.    RelationalCompare (str3, str4);
34.    return 0;
35.}
```

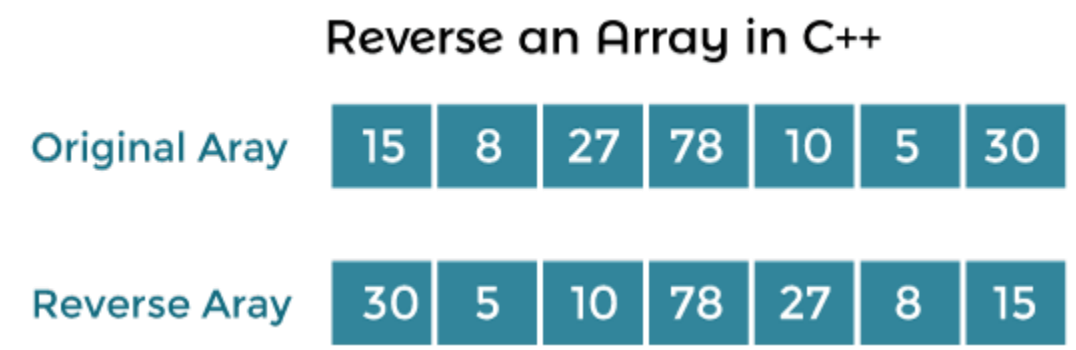
Output

```
JavaT is not equal to Tpoint string.
Tpoint is greater than JavaT string.
JavaTpoint is equal to JavaTpoint string.
```

Reverse an Array in C++

This section will discuss the different ways to reverse an array in the C++ programming language. The reverse of an array means to change the order of the given array's elements. This technique reverses the last element of the array into the first one, and the first element becomes the last. However, the process continues until all characters or elements of the array are completely reversed.

For example, the array contains elements like 'H', 'E', 'L', 'L', 'O', and when we reverse all the elements of an array, it returns the inverted array as 'O', 'L', 'L', 'E', 'H'. So, this way, all the characters in the array are reversed.



Different ways to reverse an array

Following are the various ways to get the reverse array in the C++ programming language.

- Reverse an array using for loop
- Reverse an array using the reverse() function

- Reverse an array using the user-defined function
- Reverse an array using the pointers
- Reverse an array using the Recursion function

Program to reverse an array using the for loop

Let's create a program to reverse the elements of the array using for loop in C++.

Program1.cpp

```
1. #include <iostream>
2. using namespace std;
3.
4. int main ()
5. {
6.     int arr[50], num, temp, i, j;
7.     cout << " Please, enter the total no. you want to enter: ";
8.     cin >> num;
9.
10.    // use for loop to enter the numbers
11.    for (i = 0; i < num; i++)
12.    {
13.        cout << " Enter the element " << i+1 << ": ";
14.        cin >> arr[i];
15.    }
16.
17.
18.    for ( i = 0, j = num - 1; i < num/2; i++, j--)
19.    {
20.        temp = arr[i];
21.        arr[i] = arr[j];
22.        arr[j] = temp;
23.    }
24.    cout << "\n Reverse all elements of the array: " << endl;
25.    // use for loop to print the reverse array
26.    for ( i = 0; i < num; i++)
27.    {
28.        cout << arr[i] << " ";
29.    }
30.    return 0;
31.}
```

Output

```
Please, enter the total no. you want to enter: 6
Enter the element 1: 78
Enter the element 2: 12
Enter the element 3: 54
Enter the element 4: 24
Enter the element 5: 7
Enter the element 6: 90

Reverse all elements of the array:
90 7 24 54 12 78
```

Program to reverse an array using the reverse() function

Let's consider an example to print the reverse of an array using the reverse () function in C++.

Program2.cpp

```
1. #include <iostream>
2. #include <algorithm>
3. using namespace std;
4.
```



```

5. // declare disp() function
6. void disp(int arr1[], int num)
7. {
8.     int i;
9.     // use for loop to iterate the characters
10.    for ( i = 0; i < num; i++)
11.    {
12.        cout << arr1[i] << " ";
13.    }
14. }
15.
16. // define reverse() function to reverse the array elements
17. void reverse(int arr1[], int num)
18. {
19.     reverse(arr1, arr1 + num);
20. }
21.
22. int main ()
23. {
24.     // declare and initialize an array
25.     int arr1[] = {34, 78, 21, 90, 5, 2};
26.     int num = sizeof(arr1)/sizeof(arr1[0]);
27.
28.     // call reverse function and pass parameters
29.     reverse(arr1, num);
30.     disp(arr1, num); /* call disp() function to print the revrse array. */
31.
32.     return 0;
33. }

```

Output

```
2 5 90 21 78 34
```

Program to reverse an array using the user-defined function

Let's consider an example to display the reverse of array elements using the user-defined in C++.

Program3.cpp

```

1. #include <iostream>
2. using namespace std;
3.
4. void ArrRev ( int [], int);
5. int main ()
6. {
7.     int arr[50], num, i, j, temp;
8.     cout << " Number of elements to be entered: " << endl;
9.     cin >> num;
10.
11.    cout << " Enter the array elements: " << endl;
12.
13.    // use for loop to enter the elements
14.    for ( i = 0; i < num; i++)
15.    {
16.        cin >> arr[i];
17.    }
18.    cout << " Elements are: \n";
19.    // display entered elements in array
20.    for ( i = 0; i < num; i++)

```

```

21. {
22.     cout << arr[i] << " ";
23. }
24. ArrRev (arr, num); // call function
25.
26. cout << " \n The reverse of the given array is: \n";
27. // use for loop to print the reverse array elements
28. for ( i = 0; i < num ; i++)
29. {
30.     cout << arr[i] << " ";
31. }
32. cout << endl;
33. return 0;
34. }
35.
36. void ArrRev ( int ar[], int a2)
37. {
38.     int i, j, temp;
39.     j = a2 - 1;
40.     for ( i = 0; i < j; i++, j--)
41.     {
42.         temp = ar[i];
43.         ar[i] = ar[j];
44.         ar[j] = temp;
45.     }
46. }

```

Output

```

Number of elements to be entered:
7
Enter the array elements:
45
32
89
21
78
34
65
Elements are:
45 32 89 21 78 34 65
The reverse of the given array is:
65 34 78 21 89 32 45

```

Program to reverse an array using the pointer

Let's consider an example to demonstrate the reversing of the array elements using the pointers in C++.

Program4.cpp

```

1. #include <iostream>
2. using namespace std;
3. int main ()
4. {
5.     // declare the size of array
6.     int arr[50], arr2[50];
7.     int *ptr, i, num;
8.     cout << " No. of array elements to be entered: " << endl;
9.     cin >> num;
10.
11.     cout << " Enter the elements: ";
12.     // use for loop to insert the array elements
13.     for (i = 0; i < num; i++)
14.     {
15.         cin >> arr[i];

```

```

16. }
17. ptr = &arr[0];
18. cout << " Entered elements of the array are: \n" << endl;
19. for (i = 0; i < num; i++)
20. {
21.     cout << "\t" << *ptr;
22.     ptr++;
23. }
24. ptr--; // decrement ptr
25. for ( i = 0; i < num; i++)
26. {
27.     arr2[i] = *ptr;
28.     ptr--;
29. }
30. ptr = &arr2[0];
31. for ( i = 0; i < num; i++)
32. {
33.     arr[i] = *ptr;
34.     ptr++; // increment ptr
35. }
36. ptr = &arr[0]; // ptr hold the base address of arr[0]
37. cout << "\n The reversed array elements are: \n " << endl;
38.
39. // print the array elements using ptr
40. for ( i = 0; i < num; i++)
41. {
42.     cout << "\t " << *ptr << endl;
43.     ptr++;
44. }
45. return 0;
46. }

```

Output

```

No. of array elements to be entered:
6
Enter the elements: 45
32
89
63
4
6
Entered elements of the array are:

    45    32    89    63    4    6
The reversed array elements are:

    6
    4
    63
    89
    32
    45

```

Reverse an array using the Recursion function

Let's create a program to reverse the array elements using the recursion function in C++.

Program5.cpp

```

1. #include <iostream>
2. using namespace std;
3.
4. // initialize array
5. int arr[] = { 20, 34, 5, 8, 1, 78};
6.
7. // size of the array

```

```

8.  int size = sizeof( arr) / sizeof (arr[0]);
9.
10. void reverseArr ( int arr[] , int num)
11. {
12.     // check the size of array
13.     if ( num == size)
14.         return;
15.
16.     // extract array elements
17.     int elem = arr [ num];
18.
19.     // recursively calls the next element of the array
20.     reverseArr ( arr, num + 1);
21.     // assigning elements
22.     arr [ size - num - 1] = elem;
23. }
24.
25. int main ()
26. {
27.     int i;
28.     // call recursive function (start from first elements
29.     cout << " Original elements of the arrays " << endl;
30.     for ( int i = 0; i < size; i++)
31.     {
32.         cout << arr[i] << " ";
33.     }
34.
35.     reverseArr (arr, 0);
36.     cout << " \n Reverse elements of the array are: " << endl;
37.     // display the array elements
38.     for ( int i = 0; i < size; i++)
39.     {
40.         cout << arr[i] << " ";
41.     }
42.     return 0;
43. }

```

Output

```

Original elements of the arrays
20 34 5 8 1 78
Reverse elements of the array are:
78 1 8 5 34 20

```

C++ date and time

In this article, we will learn about the date and time formats in C++. There is no complete format in C++ for date and time so we inherit it from the c language. To use date and time in c++, **<ctime>** header file is added in the program.

<ctime>

This header file has four time-related types as follows -

- **Clock_t** - It stands for clock type which is an alias of arithmetic type. It represents clock tick counts(units of a time of a constant with system-specific length). Clock_t is the type returned by **clock()/**.
- **Time_t** - It stands for time_type. It represents the time that is returned by function **time()**. It outputs an integral value as the number of seconds elapsed when time passes by 00:00 hours.
- **Size_t** - It is an alias for an unsigned integer type and represents the size of any object in bytes. Size_t is the result of the **sizeof()** operator which prints sizes and counts.
- **tm** - The tm structure holds date and time in the C structure. It is defined as follows -

- 1. `struct tm {`
- 2. `int tm_sec; // seconds of minutes from 0 to 61`
- 3. `int tm_min; // minutes of hour from 0 to 59`
- 4. `int tm_hour; // hours of day from 0 to 24`
- 5. `int tm_mday; // day of month from 1 to 31`
- 6. `int tm_mon; // month of year from 0 to 11`
- 7. `int tm_year; // year since 1900`
- 8. `int tm_wday; // days since sunday`
- 9. `int tm_yday; // days since January 1st`
- 10. `int tm_isdst; // hours of daylight savings time`
- 11. }

Date and time functions in c++

Name of the function	Prototype of the function	Description About the function
mktime	time_t mktime(struct tm *time);	This function converts mktime to time_t or calendar date and time.
ctime	char *ctime(const time_t *time);	It returns the pointer to a string of the format - day month year hours: minutes: seconds year.
difftime	double difftime (time_t time2, time_t time1);	It returns the difference of two-time objects t1 and t2.
gmtime	struct tm *gmtime(const time_t *time);	This function returns the pointer of the time in the format of a structure. The time is in UTC.
clock	clock_t clock(void);	It returns an approximated value for the amount of time the calling program is being run. The value .1 is returned if not available.
localtime	struct tm *localtime(const time_t *time);	This function returns the pointer to the tm structure representing local time.
time	time_t time(time_t *time);	It represents current time.
strftime	size_t strftime();	With the help of this function, we can format date and time in a specific manner.
asctime	char * asctime (const struct tm * time);	The function converts the type object of tm to string and returns the pointer to that string.

Example to print current date and time

Below is the example to print the current date and time in the UTC format.

Code

- 1. `#include <ctime>`

```

2. #include <iostream>
3.
4.
5. using namespace std;
6.
7. int main()
8. {
9.
10.    time_t now = time(0); // get current dat/time with respect to system
11.
12.    char* dt = ctime(&now); // convert it into string
13.
14.    cout << "The local date and time is: " << dt << endl; // print local date and time
15.
16.    tm* gmtm = gmtime(&now); // for getting time to UTC convert to struct
17.    dt = asctime(gmtm);
18.    cout << "The UTC date and time is:" << dt << endl; // print UTC date and time
19.}

```

Output

```

The local date and time is: Wed Sep 22 16:31:40 2021
The UTC date and time is: Wed Sep 22 16:31:40 2021

```

The below code tells how to break the tm structure and to print each attribute independently with the use of -> operator.

Code

```

1. #include <iostream>
2. #include <ctime>
3. using namespace std;
4.
5. int main()
6. {
7.    time_t now = time(0); // get current date and time
8.
9.    cout << "Number of seconds since January 1,2021 is:: " << now << endl;
10.
11.    tm* ltm = localtime(&now);
12.
13.    // print various components of tm structure.
14.    cout << "Year:" << 1900 + ltm->tm_year << endl; // print the year
15.    cout << "Month: " << 1 + ltm->tm_mon << endl; // print month number
16.    cout << "Day: " << ltm->tm_mday << endl; // print the day
17.    // Print time in hour:minute:second
18.    cout << "Time: " << 5 + ltm->tm_hour << ":";
19.    cout << 30 + ltm->tm_min << ":";
20.    cout << ltm->tm_sec << endl;
21.}

```

Output

```

Number of seconds since January 1,2021 is:: 1632328553
Year:2021
Month: 9
Day: 22
Time: 21:65:53

```

Copy elision in C++

Copy elision is defined as an optimisation technique that is used to avoid the unnecessary copying of objects. Generally, all the compilers use the technique of copy elision. The optimisation technique is not available to a temporary object that is bound to a reference.

It is also known as copy omission.

Let us understand the need for copy elision with the help of an example.

Code



```
1. #include <iostream>
2. using namespace std;
3. class A
4. {
5. public:
6.     A(const char* str = "\0") //default constructor
7.     {
8.         cout << " Default Constructor called" << endl;
9.     }
10.
11.     A(const A &a) //copy constructor
12.     {
13.         cout << "Copy constructor called" << endl;
14.     }
15. };
16.
17. int main()
18. {
19.     A a1 = "copy me"; // Create object of class A
20.     return 0;
21. }
```

Output

```
Default Constructor called
```

Observations

The program outputs the *Default constructor*. It happened because when we created object a1 one argument constructor is converted to copy me to a temp object and that temp object is copied to object a1.

This is how the statement -

A a1 = "copy me"

Is converted to

A a1("copy me")

How to avoid unnecessary overheads?

This problem of overhead is avoided by many compilers.

The modern compilers break down the statement of copy initialisation

A a1 = "copy me"

To,

The statement of direct initialisation.

A a1("copy me")

which in turn calls the copy constructor.

Array of sets in C++

An array is defined as the collection of data items stored in a contiguous manner. An array stores different variables of the same type. It becomes easier to access the variables as they are stored at continuous locations.

For example

10	20	30	40	50	60
0	1	2	3	4	5

This is an array that contains six elements. Let's say the array name is arr[]. Now, if we want to access the elements, they can access the indices 0 to n-1. Here, n is the size of the array.

Arr[0] = 10



Arr[1] = 20

Arr[2] = 30

Arr[3] = 40

Arr[4] = 50

Arr[5] = 60

Sets

A set is an associative container in which the elements are unique. Unlike an array, the value entered in a set cannot be modified after adding it. However, if we want to update the value in the set, we can remove it first and then enter the modified value.

Syntax

- 1. set<data_type> variable_name

Example

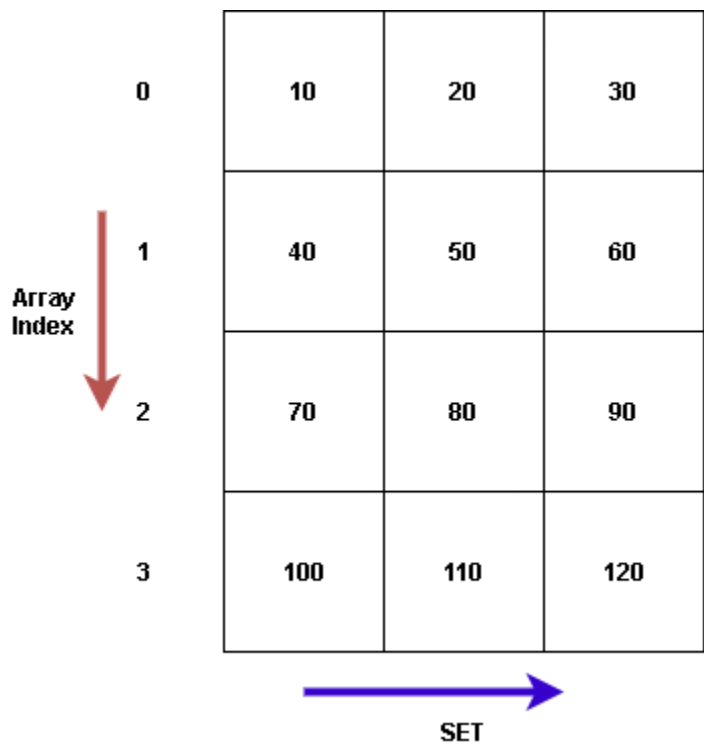
set<int> s ----- A set of integer values

set<char> c ---- A set of character values

Array of sets

When we say an array of sets, it is a two-dimensional array with a fixed number of rows. Each row can have variable lengths.

In an array of sets, each array index stores a set. The set is accessible via iterators. \



Syntax

- 1. set<data_type> variable_name[size_of_array]

Example

set<int> s[3] ----> A array of set of int type with size 3

Insertion operation in the array of sets

Insertion will happen with the insert() function as each row is a set here. So, we insert using an array of sets using -

set_variable[row_number].insert(element)

Example

s[0].insert(10) ----> Inserts 10 in row 1

s[1].insert(20) ----> Inserts 20 in row 2

Code

- 1. #include <bits/stdc++.h>
- 2. using namespace std;
- 3. #define ROW 3
- 4. #define COL 4
- 5.
- 6. int main()
- 7. {
- 8.
- 9. set<int> s[ROW]; // Create an array of sets
- 10.

```

11.  int num = 10; // Initial element to be inserted
12.
13.  for (int i = 0; i < ROW; i++) { // iterate in row
14.      // Insert the column elements
15.      for (int j = 0; j < COL; j++) {
16.          s[i].insert(num);
17.          num += 10;
18.      }
19.  }
20.
21.  // Display the array of sets
22.  for (int i = 0; i < ROW; i++) {
23.      cout << "Row " << i + 1 << " = "
24.          << "Elements at index " << i << ": ";
25.
26.      // Print the array of sets
27.      for (auto x : s[i])
28.          cout << x << " ";
29.
30.      cout << endl;
31.  }
32.
33.  return 0;
34. }

```

Output

```

Row 1 = Elements at index 0: 10 20 30 40
Row 2 = Elements at index 1: 50 60 70 80
Row 3 = Elements at index 2: 90 100 110 120

```

Deletion operation in an array of sets

When we say deletion of an element here, we are removing the element from the set. The function to remove an element from the set is **erase()**.

Code

In this example, we have removed one element from set 3 and one element from set 2. Remember, the set index is i-1 as the array index starts at 0.

```

1.  #include <bits/stdc++.h>
2.  using namespace std;
3.
4.  // Fixing rows and column
5.  #define ROW 3
6.  #define COL 4
7.
8.  int main()
9.  {
10.   set<int> s[ROW]; // Create array of set
11.   int num = 10;
12.
13.   // Put elements in the set at a increment of 10
14.   for (int i = 0; i < ROW; i++) {
15.
16.       // insert the column elements
17.       for (int j = 0; j < COL; j++) {
18.           s[i].insert(num);
19.           num += 10;
20.       }

```

```

21. }
22.
23. cout << "Before removal elements are:"
24.     << endl;
25.
26. // Print elements after insertion
27. for (int i = 0; i < ROW; i++) {
28.     cout << "Elements at index "
29.         << i << ": ";
30.     for (auto x : s[i])
31.         cout << x << " ";
32.     cout << endl;
33. }
34.
35. s[2].erase(100); // Erase 100 from row 3 which means set 3
36.
37. s[1].erase(50); // Erase 50 from row 2 which means set 2
38.
39. cout << endl
40.     << "After removal elements are:" << endl;
41.
42. for (int i = 0; i < ROW; i++) {
43.     cout << "Elements at index " << i << ": ";
44.
45.     // Print the current set
46.     for (auto x : s[i])
47.         cout << x << " ";
48.
49.     cout << endl;
50. }
51.
52. return 0;
53.}

```

Output

```

Before removal elements are:
Elements at index 0: 10 20 30 40
Elements at index 1: 50 60 70 80
Elements at index 2: 90 100 110 120

After removal elements are:
Elements at index 0: 10 20 30 40
Elements at index 1: 60 70 80
Elements at index 2: 90 110 120

```

Traversal operation in an array of sets

When we traverse an array of sets, we iterate through each set and print all the elements in that set. Iterators are used to traverse the set elements.

Code

In the below example, we create an array of sets having two rows. Row one has three elements in the set, and row 2 has two elements in the set.

To traverse an array of sets, we run an outer loop for the rows. In the inner loop, we use iterators to print each set.

```

1. #include <bits/stdc++.h>
2. using namespace std;
3. #define ROW 2 // Making a set of 2 rows
4. int main()
5. {
6.

```

```

7.   set<int> s[ROW]; // create an array of sets with 2 rows
8.
9.   // Insertion in row 1
10.  s[0].insert(10); // insertion in column 1
11.  s[0].insert(15); // insertion in column 2
12.  s[0].insert(35); // insertion in column 3
13.
14.  // Insertion in row 2
15.  s[1].insert(20); // insertion in column 1
16.  s[1].insert(30); // insertion in column 2
17.
18.  // Traversal in array of sets
19.  for (int i = 0; i < ROW; i++) { // Iterate in the row
20.      cout << "Elements at index " << i << ": ";
21.      // Now we have to Iterate in the set of a particular row
22.      // So use an Iterate that runs from begin of set to end of that set
23.      for (auto it = s[i].begin(); it != s[i].end(); it++) {
24.
25.          // Print the set value
26.
27.          cout << *it << ' ';
28.      }
29.
30.      cout << endl;
31.  }
32.
33.  return 0;
34. }

```

Output

```

Elements at index 0: 10 15 35
Elements at index 1: 20 30

```

Smart pointers in C++

A pointer is used to store the address of another variable. In other words, a pointer extracts the information of a resource that is outside the program (heap memory).

We use a copy of the resource, and to make a change, it is done in the copied version. To change the content of the original resource, smart pointers are used.

Problems faced with normal pointers

Below is an example that depicts the problem with normal pointers.

Created a class **Rectangle** with two data members (*length and breadth*). A function **fun()** dynamically creates an object of Rectangle.

When the function fun() ends, the object p gets destroyed. Since we did not delete p, the memory remains allocated, and this allocated resource will not be available to other variables.

In the **main()**, we execute an infinite loop that will keep creating p objects and allocate resources. This problem turns out into a **memory leak** to which smart pointers is a solution.

Code

```

1.  #include <iostream>
2.  using namespace std;
3.
4.  class Rectangle { // created a class Rectangle
5.  private:

```

```

6.   int length; // data member as length of rectangle
7.   int breadth; // data member as breadth of rectangle
8.
9. };
10.
11. void fun() // the function to indicate the problem with normal pointer
12. {
13.
14.   Rectangle* p = new Rectangle(); // Create a dynamic object p
15. }
16.
17. int main()
18. {
19.   // Infinite Loop
20.   while (1) { // Run an infinite loop that will allocate p
21.     fun();
22.   }
23. }

```

Note - The languages such as JAVA, C# has their garbage collectors that smartly deallocate unused memory resource.

Smart pointers

We will implement smart pointers such that they will release the memory of unused resources.

Create a class with a pointer, overloaded operators(->, *) and destructors.

The destructor will be automatically called when its object goes out of the scope, and automatically the dynamically allocated memory will be deleted.

Example

```

1. #include <iostream>
2. using namespace std;
3.
4. class SmartPtr { // Create the class to implement smart Pointer
5.   int* ptr; // Actual pointer
6. public:
7.   // Create an explicit constructor
8.   explicit SmartPtr(int* p = NULL) { ptr = p; }
9.
10.  // Destructor to deallocate the resource used
11.  ~SmartPtr() { delete (ptr); }
12.
13.  // Overloading dereferencing operator
14.  int& operator*() { return *ptr; }
15. };
16.
17. int main()
18. {
19.   SmartPtr ptr(new int());
20.   *ptr = 100;
21.   cout << *ptr;
22.
23.   // We don't need to call delete ptr: when the object
24.   // ptr goes out of scope, the destructor for it is automatically
25.   // called, and destructor does delete ptr.
26.
27.   return 0;
28. }

```

Output

100

The above example works only for *int*. We will create a template that will work for every data type.

Code

```
1. #include <iostream>
2. using namespace std;
3.
4. template <class T> // Create a template class
5. class SmartPtr {
6.     T* ptr; // Actual pointer
7. public:
8.     // Constructor
9.     explicit SmartPtr(T* p = NULL) { ptr = p; }
10.
11.    // Destructor
12.    ~SmartPtr() { delete (ptr); }
13.
14.    // Overloading dereferncing operator
15.    T& operator*() { return *ptr; }
16.
17.    // Overloading arrow operator so that
18.    // members of T can be accessed
19.    // like a pointer (useful if T represents
20.    // a class or struct or union type)
21.    T* operator->() { return ptr; }
22. };
23.
24. int main()
25. {
26.     SmartPtr<int> ptr(new int());
27.     *ptr = 100;
28.     cout << *ptr;
29.     return 0;
30. }
```

Output

100

Types of smart pointers

- **Unique_ptr**

This type of object stores only a single object. To assign a different object, current object is deallocated.

Code

```
1. #include <iostream>
2. #include <memory>
3. using namespace std;
4.
5.
6. class Rectangle { // Create the class
7.     // Data members
8.     int length; // length of rectangle
9.     int breadth; // breadth of rectangle
10. }
```

```

11. public:
12.     Rectangle(int l, int b)
13.     { // parameterised constructor
14.         length = l;
15.         breadth = b;
16.     }
17.
18.     int area()
19.     { // calculate area
20.         return length * breadth; // return the area
21.     }
22. };
23.
24. int main()
25. {
26.
27.     unique_ptr<Rectangle> P1(new Rectangle(20, 5));
28.     cout << P1->area() << endl; // This'll print 100
29.
30.     // unique_ptr<Rectangle> P2(P1);
31.     unique_ptr<Rectangle> P2;
32.     P2 = move(P1);
33.
34.     // This'll print 100
35.     cout << P2->area() << endl;
36.
37.     return 0;
38. }

```

Output

```

100
100

```

- **Shared_ptr**

In shared_ptr, more than one object can point to a single pointer at the same instance of time. A reference counter is maintained for denoting the object using the **use_count()** method.

Code

```

1. #include <iostream>
2. #include <memory>
3. using namespace std;
4.
5. class Rectangle { // Create the class
6.     // Data members
7.     int length; // length of rectangle
8.     int breadth; // breadth of rectangle
9.
10. public:
11.     Rectangle(int l, int b)
12.     { // parameterised constructor
13.         length = l;
14.         breadth = b;
15.     }
16.
17.     int area()
18.     { // calculate area of rectangle
19.         return length * breadth; // return area
20.     }

```

```

21. };
22.
23. int main()
24. {
25.
26.     shared_ptr<Rectangle> P1(new Rectangle(20, 5)); // create shared //ptr P1
27.     // This'll print 100
28.     cout << P1->area() << endl;
29.     // Create shared ptr P2
30.     shared_ptr<Rectangle> P2;
31.     P2 = P1;
32.
33.     // This'll print 100
34.     cout << P2->area() << endl;
35.
36.     // This'll now not give an error,
37.     cout << P1->area() << endl; // prints 100
38.
39. // reference counter of P2 is 2
40.     cout << P1.use_count() << endl; // prints 2
41.     return 0;
42. }

```

Output

```

100
100
100
2

```

- **Weak_ptr**

Weak_ptr is similar to the shared pointer. The difference is that it does not maintain a reference counter and there is no strong hold of the object on the pointer. This property may result in a deadlock as different objects will try to hold the pointer.

Types of polymorphism in C++

Polymorphism is defined as the process of using a function or an operator for more than one purpose. In other words, we can also say that an operator or a function can serve us in different ways.

For example

Let us say an operator '+' is used to add two integer numbers and it is also used to concatenate two strings.

Hence the '+' operator serves two purposes - adding and concatenation.

Now let us learn about types of polymorphism in C++.

Compile-time polymorphism

It is defined as the polymorphism in which the function is called at the compile time. This type of process is also called as *early or static binding*.

Examples of compile-time polymorphism

1) Function overloading

Function overloading is defined as using one function for different purposes. Here, one function performs many tasks by changing the function signature(number of arguments and types of arguments). It is an example of compile-time polymorphism because what function is to be called is decided at the time of compilation.

Example

In the below code, we have taken a **class Addition** that performs the addition of two numbers and concatenates two strings. It has a function **ADD()** that is overloaded two times.

The function **ADD(int X, int Y)** adds two integers and the function **ADD()** with no arguments performs string concatenation.

Code

```
1. #include <iostream>
2. using namespace std;
3. class Addition { // Create Addition class
4. public:
5.     int ADD(int X, int Y) // Add function add two numbers
6.     {
7.         return X + Y; // return the Addition of two numbers
8.     }
9.     int ADD()
10.    { // this ADD function concatenates two strings
11.        string a = "HELLO";
12.        string b = " JAVATPOINT";
13.        string c = a + b; // concatenate two strings
14.        cout << c << endl; // print result
15.    }
16. };
17. int main(void)
18. {
19.     Addition obj; // Object is created of Addition class
20.     cout << obj.ADD(120, 105) << endl; //first method is called to add two integers
21.     obj.ADD(); // second method is called to concatenate two strings
22.     return 0;
23. }
```

Output

```
225
HELLO JAVATPOINT
```

Operator Overloading

Operator overloading is defined as using an operator for an addition operation besides the original one.

The concept of operator overloading is used as it provides special meanings to the user-defined data types. The benefit of operator overloading is we can use the same operand to serve two different operations. The basic operator overloading example is the '+' operator as it is used to add numbers and strings.

The operators , :: ?: **sizeof** cant be overloaded.

Example

In the below example, we have overloaded '+' to add two strings.

An operator is overloaded by using the **operator** keyword with the operator to be overloaded.

Code

```
1. #include <bits/stdc++.h>
2. using namespace std;
3. class A // Create a class A
4. {
5.
6.     string x; // private data member x
7. public:
8.     A() {}
9.     A(string i)
10.    {
11.        x = i; // Assign the data member
12.    }
```

```

13. void operator+(A); // operator overloading
14. };
15.
16. void A::operator+(A a) // concetenate the strings and print output
17. {
18.
19.     string m = x + a.x;
20.     cout << "The result of the addition of two objects is: " << m;
21. }
22. int main()
23. {
24.     A a1("Welcome "); // string 1
25.     A a2("to javatpoint"); // string 2
26.     a1 + a2; // concetenate two strings using operator overloading
27.     return 0;
28. }

```

Output

```
The result of the addition of two objects is: Welcome to javatpoint
```

Run-time polymorphism

The run-time polymorphism is defined as the process in which the function is called at the time of program execution.

An example of runtime polymorphism is *function overriding*.

Function overriding

When we say a function is overridden it means that a new definition of that function is given in the derived class. Thus, in function overriding, we have two definitions of one in the base class and the other in the derived class. The decision of choosing a function is decided at the run time.

Example

In the below example, a class animal has a function **f()** which is also in the derived class **Man**. When we call the function f() with the base object, it prints the contents of the base and with the derived object the content of the derived. Thus, choosing the function f() is decided on runtime.

Code

```

1. #include <iostream>
2. using namespace std;
3. class Animal { // Create a new class
4. public:
5.     void f()
6.     { // the function f will be overridden in the class Man
7.         cout << "Eating..." << endl;
8.     }
9. };
10. class Man : public Animal // Base class inheriting derived class
11. {
12. public:
13.     void f() // The function f being overridden
14.     {
15.         cout << "Walking ..." << endl;
16.     }
17. };
18. int main(void)
19. {
20.     Animal A = Animal();
21.     A.f(); //parent class object

```

```

22.   Man m = Man();
23.   m.f(); // child class object
24.
25.   return 0;
26. }

```

Output

```

Eating...
Walking ...

```

Implementing the sets without C++ STL containers

A set is defined as a collection of elements where each element is unique. It is different from arrays as sets can have variable lengths. The element added to a set once cannot be changed. If we want to add the same modified number, delete the number and add the modified one in the set.

Following are the operations that can be performed on a set -

- **add(value)** - This function adds an element in the set
- **intersectionSet(S)** - Return the intersection of two sets in the set S
- **toArray()** - Converts a set into an array of elements
- **contains(s)** - Returns the set if an element to be searched is present in the set
- **displaySet()** - Print the set from beginning to end
- **getSize()** - To get the size of the set
- **unionSet(s)** - To display the union of two sets with set s

Implementation of sets using BST

The set data structure internally implements BST (Binary Search Tree) data structure. Hence, we will add the elements in the tree and use this tree template to implement the Set.

- **Create a template of BST**

In a tree, we have three members - the data of the node and the left and right pointers to the node. It is given as below -

```

1. template <typename T>
2. struct Node { // Create a node of BST
3.     T data; // Node value
4.     Node* left; // Pointer to the left child
5.     Node* right; // Pointer to the right child
6. };

```

After creating a tree, we insert the nodes in the tree using the **insert()** function. In a BST, the data that is less than root lies on the left side of the tree and the greater one lies on the right side of the tree.

The Function **containsNode()** used are for checking whether a node is present in the tree or not.

The function **inorder()** is used to print the inorder traversal of the BST.

- **Implement the BST template in the Set class**

After we have created the BST for the internal working of the set, a set template is created to implement the BST. It has a root pointer node to store the data and the size variable returns the size of the set.

The Set class has a default constructor to initialize the root of BST as NULL and a copy constructor to copy a set into another set.

```

1. // create a class template for implementing a set in BST
2. template <typename T>
3. class Set { // Create class set
4.     Node<T>* root; // Root to store the data
5.     int size; // It indicates the size of the set

```

6. };

The function **add()** is used to add values in the set. It does not add the duplicate data in the set by calling the function **containsNode()**. If there is a new element it adds to the set.

The function **contains()** checks if a particular element is present in the set or not. It internally calls **containsNode()** in the BST.

The function **displaySet()** is used to print the set elements. It internally calls the **inorder()** function of the BST.

The function **getSize()** returns the size of the set.

Code

```
1. #include <algorithm>
2. #include <iostream>
3. #include <math.h>
4. #include <stack>
5. #include <string>
6. using namespace std;
7.
8. template <typename T>
9. struct Node { // Create a node of BST
10.
11.     T data; // Node value
12.
13.     Node* left; // Pointer to the left child
14.
15.     Node* right; // Pointer to the right child
16.
17. public:
18.     // this function prints inorder traversal of BST
19.     void inorder(Node* r)
20.     {
21.         if (r == NULL) { // If we reach last level
22.             return;
23.         }
24.         inorder(r->left); // print left child
25.         cout << r->data << " "; // print the node value
26.         inorder(r->right); // print the right child
27.     }
28.
29. /*
30.     Function to check if BST contains a node
31.     with the given data
32.
33.     r pointer to the root node
34.     d the data to search in the BST
35.     The function returns 1 if the node is present in the BST else 0
36. */
37. int containsNode(Node* r, T d)
38. {
39.     if (r == NULL) { // If we reach last level or tree is empty
40.         return 0;
41.     }
42.     int x = r->data == d ? 1 : 0; // Check for duplicacy
43.     // Traverse in right and left subtree
44.     return x | containsNode(r->left, d) | containsNode(r->right, d);
45. }
46.
```

```

47.  /*
48.     Function to insert a node with
49.     given data into the BST
50.
51.     r pointer to the root node in the BST
52.     d the data to insert in the BST
53.     return pointer to the root of the resultant BST
54.  */
55.  Node* insert(Node* r, T d)
56.  {
57.
58.      if (r == NULL) { // Add where NULL is encountered which means space is present
59.          Node<T>* tmp = new Node<T>; // Create a new node in BST
60.          tmp->data = d; // insert the data in the BST
61.          tmp->left = tmp->right = NULL; // Allocate left and right pointers a NULL
62.          return tmp; // return current node
63.      }
64.
65.      // Insert the node in the left subtree if data is less than current node data
66.      if (d < r->data) {
67.          r->left = insert(r->left, d);
68.          return r;
69.      }
70.
71.      // Insert the node in the right subtree if data is greater than current node data
72.      else if (d > r->data) {
73.          r->right = insert(r->right, d);
74.          return r;
75.      }
76.      else
77.          return r;
78.  }
79. };
80.
81. template <typename T> // create a class template for implementing a set in BST
82. class Set { // Create class set
83.
84.     Node<T>* root; // Root to store the data
85.
86.     int size; // It indicates the size of the set
87.
88. public:
89.     Set() // If no value passed
90.     {
91.         root = NULL; // It points to NULL
92.         size = 0; // size becomes zero
93.     }
94.
95.     Set(const Set& s) // Copy constructor
96.     {
97.         root = s.root;
98.         size = s.size;
99.     }
100.
101.     void add(const T data) // It add an element to the set
102.     {
103.         if (!root->containsNode(root, data)) { // Check if it is the first element

```

```

104.         root = root->insert(root, data); // Insert the data into the set
105.         size++; // Increment the size of the set
106.     }
107. }
108.
109. bool contains(T data)
110. {
111.     return root->containsNode(root, data) ? true : false;
112. }
113.
114. void displaySet()
115. {
116.     cout << "{ ";
117.     root->inorder(root);
118.     cout << "}" << endl;
119. }
120.
121. /*
122.     Function to return the current size of the Set
123.
124.     @return size of set
125. */
126. int getSize()
127. {
128.     return size;
129. }
130. };
131.
132. int main()
133. {
134.
135.     // Create Set A
136.     Set<int> A;
137.
138.     // Add elements to Set A
139.     A.add(1);
140.     A.add(2);
141.     A.add(3);
142.     A.add(2);
143.
144.     // Display the contents of Set A
145.     cout << "A = ";
146.     A.displaySet();
147.
148.     // Check if Set A contains some elements
149.     cout << "A " << (A.contains(3) ? "contains"
150.         : "does not contain")
151.         << " 3" << endl;
152.     cout << "A " << (A.contains(4) ? "contains"
153.         : "does not contain")
154.         << " 4" << endl;
155.     cout << endl;
156.
157.     return 0;
158. }

```

Output

```
A = { 1 2 3 }  
A contains 3  
A does not contain 4
```

Scope Resolution Operator in C++

This section will discuss the scope resolution operator and its various uses in the C++ programming language. The scope resolution operator is used to reference the global variable or member function that is out of scope. Therefore, we use the scope resolution operator to access the hidden variable or function of a program. The operator is represented as the double colon (::) symbol.



For example, when the global and local variable or function has the same name in a program, and when we call the variable, by default it only accesses the inner or local variable without calling the global variable. In this way, it hides the global variable or function. To overcome this situation, we use the scope resolution operator to fetch a program's hidden variable or function.

Uses of the scope resolution Operator

1. It is used to access the hidden variables or member functions of a program.
2. It defines the member function outside of the class using the scope resolution.
3. It is used to access the static variable and static function of a class.
4. The scope resolution operator is used to override function in the Inheritance.

Program to access the hidden value using the scope resolution (::) operator

Program1.cpp

```
1. #include <iostream>  
2. using namespace std;  
3. // declare global variable  
4. int num = 50;  
5. int main ()  
6. {  
7. // declare local variable  
8. int num = 100;  
9.  
10. // print the value of the variables  
11. cout << " The value of the local variable num: " << num;  
12.  
13. // use scope resolution operator (::) to access the global variable  
14. cout << "\n The value of the global variable num: " << ::num;  
15. return 0;  
16. }
```

Output

```
The value of the local variable num: 100  
The value of the global variable num: 50
```

Program to define the member function outside of the class using the scope resolution (::) operator

Program2.cpp

```

1. #include <iostream>
2. using namespace std;
3. class Operate
4. {
5. public:
6.     // declaration of the member function
7.     void fun();
8. };
9. // define the member function outside the class.
10. void Operate::fun() /* return_type Class_Name::function_name */
11. {
12. cout << " It is the member function of the class. ";
13. }
14. int main ()
15. {
16. // create an object of the class Operate
17. Operate op;
18. op.fun();
19. return 0;
20. }

```

Output

```
It is the member function of the class.
```

Program to demonstrate the standard namespace using the scope resolution (::) operator

Program3.cpp

```

1. #include <iostream>
2. int main ()
3. {
4. int num = 0;
5.
6. // use scope resolution operator with std namespace
7. std :: cout << " Enter the value of num: ";
8. std::cin >> num;
9. std:: cout << " The value of num is: " << num;
10. }

```

Output

```
Enter the value of num: 50
The value of num is: 50
```

Program to access the static variable using the scope resolution (::) operator

Program4.cpp

```

1. #include <iostream>
2. using namespace std;
3. class Parent
4. {
5. static int n1;
6. public:
7. static int n2;
8. // The class member can be accessed using the scope resolution operator.
9. void fun1 ( int n1)
10. {
11. // n1 is accessed by the scope resolution operator (:: )
12. cout << " The value of the static integer n1: " << Parent::n1;
13. cout << "\n The value of the local variable n1: " << n1;

```



```

14. }
15. };
16. // define a static member explicitly using :: operator
17. int Parent::n1 = 5; // declare the value of the variable n1
18. int Parent::n2 = 10;
19. int main ()
20. {
21. Parent b;
22. int n1 = 15;
23. b.fun1 (n1);
24. cout << " \n The value of the Base::n2 = " << Parent::n2;
25. return 0;
26. }

```

Output

```

The value of the static integer n1: 5
The value of the local variable n1: 15
The value of the Base::n2 = 10

```

Program to access the static member function using the scope resolution (::) operator

Program5.cpp

```

1. #include <iostream>
2. using namespace std;
3. class ABC
4. {
5. public:
6. // declare static member function
7. static int fun()
8. {
9. cout << " \n Use scope resolution operator to access the static member. ";
10. }
11. };
12. int main ()
13. {
14. // class_name :: function name
15. ABC :: fun ();
16. return 0;
17. }

```

Output

```

Use scope resolution operator to access the static member.

```

Program to override the member function using the scope resolution (::) operator

Program5.cpp

```

1. #include <iostream>
2. using namespace std;
3. class ABC
4. {
5. // declare access specifier
6. public:
7. void test ()
8. {
9. cout << " \n It is the test() function of the ABC class. ";
10. }
11. };
12. // derive the functionality or member function of the base class
13. class child : public ABC

```

```

14. {
15. public:
16. void test()
17. {
18. ABC::test();
19. cout << " \n It is the test() function of the child class. ";
20. }
21. };
22. int main ()
23. {
24. // create object of the derived class
25. child ch;
26. ch.test();
27. return 0;
28. }

```

Output

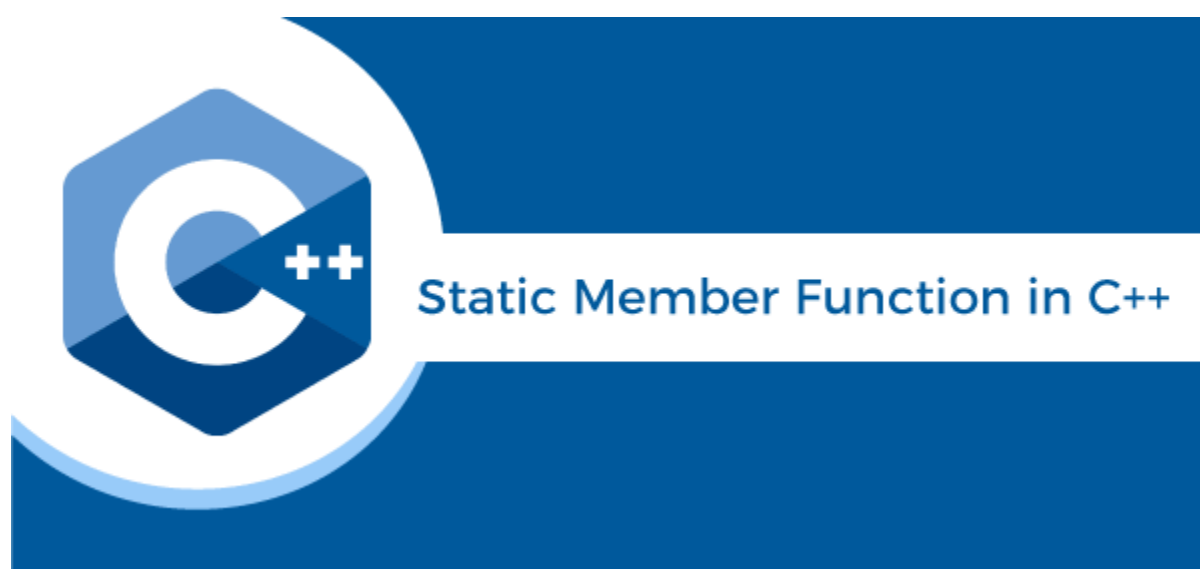
```

It is the test() function of the ABC class.
It is the test() function of the child class.

```

Static Member Function in C++

The static is a keyword in the C and C++ programming language. We use the static keyword to define the static data member or static member function inside and outside of the class. Let's understand the static data member and static member function using the programs.



Static data member

When we define the data member of a class using the static keyword, the data members are called the static data member. A static data member is similar to the static member function because the static data can only be accessed using the static data member or static member function. And, all the objects of the class share the same copy of the static member to access the static data.

Syntax

1. **static** data_type data_member;

Here, the **static** is a keyword of the predefined library.

Keep Watching

The **data_type** is the variable type in C++, such as int, float, string, etc.

The **data_member** is the name of the static data.

Example 1: Let's create a simple program to access the static data members in the C++ programming language.

1. `#include <iostream>`
2. `#include <string.h>`

```
3. using namespace std;
4. // create class of the Car
5. class Car
6. {
7. private:
8. int car_id;
9. char car_name[20];
10. int marks;
11.
12. public:
13. // declare a static data member
14. static int static_member;
15.
16. Car()
17. {
18. static_member++;
19. }
20.
21. void inp()
22. {
23. cout << " \n\n Enter the Id of the Car: " << endl;
24. cin >> car_id; // input the id
25. cout << " Enter the name of the Car: " << endl;
26. cin >> car_name;
27. cout << " Number of the Marks (1 - 10): " << endl;
28. cin >> marks;
29. }
30.
31. // display the entered details
32. void disp ()
33. {
34. cout << " \n Id of the Car: " << car_id;
35. cout << " \n Name of the Car: " << car_name;
36. cout << " \n Marks: " << marks;
37.
38. }
39. };
40.
41. // initialized the static data member to 0
42. int Car::static_member = 0;
43.
44. int main ()
45. {
46. // create object for the class Car
47. Car c1;
48. // call inp() function to insert values
49. c1. inp ();
50. c1. disp();
51.
52. //create another object
53. Car c2;
54. // call inp() function to insert values
55. c2. inp ();
56. c2. disp();
57.
58.
59. cout << " \n No. of objects created in the class: " << Car :: static_member << endl;
```

```
60. return 0;
61.}
```

Output

```
Enter the Id of the Car:
101
Enter the name of the Car:
Ferrari
Number of the Marks (1 - 10):
10

Id of the Car: 101
Name of the Car: Ferrari
Marks: 10

Enter the Id of the Car:
205
Enter the name of the Car:
Mercedes
Number of the Marks (1 - 10):
9

Id of the Car: 205
Name of the Car: Mercedes
Marks: 9
No. of objects created in the class: 2
```

Static Member Functions

The static member functions are special functions used to access the static data members or other static member functions. A member function is defined using the static keyword. A static member function shares the single copy of the member function to any number of the class' objects. We can access the static member function using the class name or class' objects. If the static member function accesses any non-static data member or non-static member function, it throws an error.

Syntax

1. `class_name::function_name (parameter);`

Here, the **class_name** is the name of the class.

function_name: The function name is the name of the static member function.

parameter: It defines the name of the pass arguments to the static member function.

Example 2: Let's create another program to access the static member function using the class name in the C++ programming language.

```
1. #include <iostream>
2. using namespace std;
3. class Note
4. {
5. // declare a static data member
6. static int num;
7.
8. public:
9. // create static member function
10. static int func ()
11. {
12. return num;
13. }
14. };
15. // initialize the static data member using the class name and the scope resolution operator
16. int Note :: num = 5;
17.
18. int main ()
19. {
20. // access static member function using the class name and the scope resolution
21. cout << " The value of the num is: " << Note:: func () << endl;
22. return 0;
23. }
```

Output

```
The value of the num is: 5
```

Example 3: Let's create another program to access the static member function using the class' object in the C++ programming language.

```
1. #include <iostream>
2. using namespace std;
3. class Note
4. {
5. // declare a static data member
6. static int num;
7.
8. public:
9. // create static member function
10. static int func ()
11. {
12. cout << " The value of the num is: " << num << endl;
13. }
14. };
15. // initialize the static data member using the class name and the scope resolution operator
16. int Note :: num = 15;
17.
18. int main ()
19. {
20. // create an object of the class Note
21. Note n;
22. // access static member function using the object
23. n.func();
24.
25. return 0;
26. }
```

Output

```
The value of the num is: 15
```

Example 4: Let's consider an example to access the static member function using the object and class in the C++ programming language.

```
1. #include <iostream>
2. using namespace std;
3. class Member
4. {
5.
6. private:
7. // declaration of the static data members
8. static int A;
9. static int B;
10. static int C;
11.
12. // declare public access specifier
13. public:
14. // define the static member function
15. static void disp ()
16. {
17. cout << " The value of the A is: " << A << endl;
18. cout << " The value of the B is: " << B << endl;
19. cout << " The value of the C is: " << C << endl;
20. }
```

```

21. };
22. // initialization of the static data members
23. int Member :: A = 20;
24. int Member :: B = 30;
25. int Member :: C = 40;
26.
27. int main ()
28. {
29. // create object of the class Member
30. Member mb;
31. // access the static member function using the class object name
32. cout << " Print the static member through object name: " << endl;
33. mb. disp();
34. // access the static member function using the class name
35. cout << " Print the static member through the class name: " << endl;
36. Member::disp();
37. return 0;
38. }

```

Output

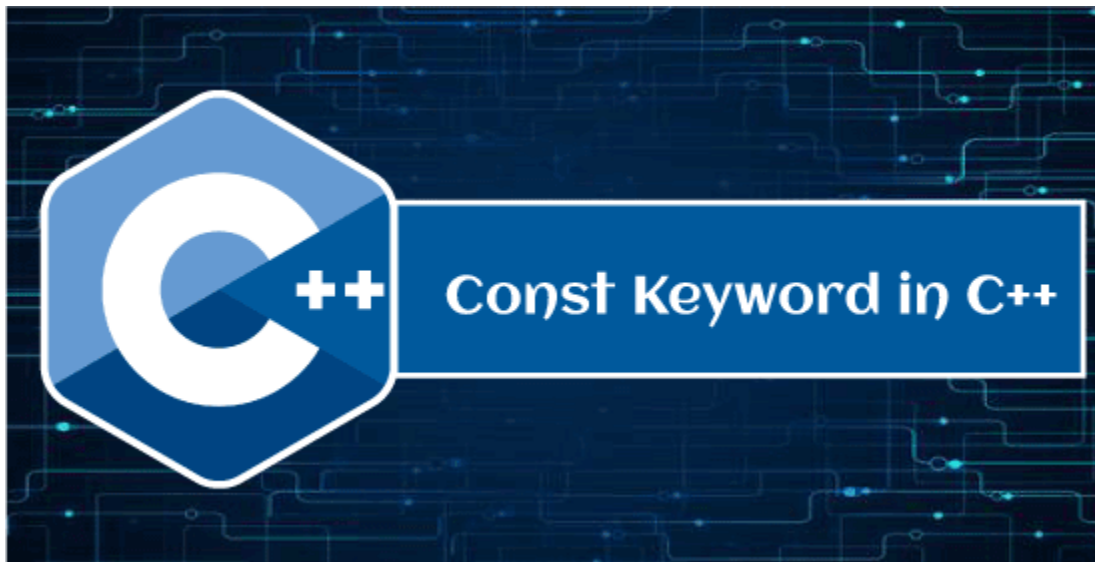
```

Print the static member through object name:
The value of the A is: 20
The value of the B is: 30
The value of the C is: 40
Print the static member through the class name:
The value of the A is: 20
The value of the B is: 30
The value of the C is: 40

```

Const keyword in C++

This section will discuss the const keyword in the C++ programming language. It is the const keywords used to define the constant value that cannot change during program execution. It means once we declare a variable as the constant in a program, the variable's value will be fixed and never be changed. If we try to change the value of the const type variable, it shows an error message in the program.



Use const keywords with different parameters:

- Use const variable
- Use const with pointers
- Use const pointer with variables
- Use const with function arguments
- Use const with class member functions
- Use const with class data members
- Use const with class objects

1. Const variable

It is a const variable used to define the variable values that never be changed during the execution of a program. And if we try to modify the value, it throws an error.

Syntax

1. **const** data_type variable_name;

Example: Program to use the const keyword in C++

Let's create a program to demonstrate the use of the const keyword in the C++ programming language.

```
1. #include <iostream>
2. #include <conio.h>
3. using namespace std;
4. int main ()
5. {
6. // declare the value of the const
7. const int num = 25;
8. num = num + 10;
9. return 0;
10. }
```

Output

It shows the compile-time error because we update the assigned value of the num 25 by 10.

Example: Let's create a simple program to demonstrate the use of the const keyword in the C++ programming language.

```
1. /* create a program to use the const keyword with different data types in the C++. */
2. #include <iostream>
3. #include <conio.h>
4. using namespace std;
5. int main ()
6. {
7. // declare variables
8. const int x = 20;
9. const int y = 25;
10. int z;
11. // add the value of x and y
12. z = x + y;
13. cout << " The sum of the x and y is: " << z << endl;
14. return 0;
15. }
```

Output

```
The sum of the x and y is: 42
```

In the above program, we declared and assigned the const variables x and y at initial. And then, store the result of two const variables to the 'z' variable to print the sum.

Note: While the declaration of the const variable in the C++ programming, we need to assign the value of the defined variables at the same time; else, it shows the compile-time error.

2. Constant pointer

To create a const pointer, we need to use the const keyword before the pointer's name. We cannot change the address of the const pointer after its initialization, which means the pointer will always point to the same address once the pointer is initialized as the const pointer.

Example: Program to demonstrate the constant pointer using the const keyword

Let's consider an example to use the const keyword with the constant pointer in the C++ programming language.

```
1. #include <iostream>
2. using namespace std;
3. int main ()
4. {
5. // declaration of the integer variables
6. int x = 10, y = 20;
7.
8. // use const keyword to make constant pointer
9. int* const ptr = &x; // const integer ptr variable point address to the variable x
10.
11. // ptr = &y; // now ptr cannot changed their address
12. *ptr = 15; // ptr can only change the value
13. cout << " The value of x: " << x << endl;
14. cout << " The value of ptr: " << *ptr << endl;
15. return 0;
16. }
```

Output

```
The value of x: 15
The value of ptr: 15
```

In the above program, pointer ptr pointing to the address of int variable 'x' and ptr variable cannot be changed their address once it initialized, but the pointer ptr can change the value of x.

3. Pointer to constant variable

It means the pointer points to the value of a const variable that cannot change.

Declaration of the pointer to the constant variable:

```
1. const int* x;
```

Here, x is a pointer that point to a const integer type variable, and we can also declare a pointer to the constant variable as,

```
1. char const* y;
```

In this case, y is a pointer to point a char type const variable.

Example: Program to use the const keyword with a pointer to constant variable

```
1. #include <iostream>
2. using namespace std;
3. int main ()
4. {
5. // declare integer variable
6. int x = 7, y = 10;
7.
8. const int *ptr = &x; // here x become constant variable
9. cout << " \n The initial value of ptr:" << *ptr;
10. cout << " \n The value of x: " << x;
11.
12. // *ptr = 15; It is invalid; we cannot directly assign a value to the ptr variable
13. ptr = &y; // here ptr variable pointing to the non const address 'y'
14.
15. cout << " \n The value of y: " << y;
16. cout << " \n The value of ptr:" << *ptr;
17. return 0;
18. }
```

Output

```
The initial value of ptr: 7
The value of x: 7
```



```
The value of y: 10
The value of ptr: 10
```

In the above program, pointer ptr points to the const int (x) variable, and the value of the int (x) variable can never change.

4. Constant function Arguments

We can declare the function arguments as the constant argument using the const keyword. And if the value of function arguments is declared const, it does not allow changing its value.

Syntax

1. return_type fun_name (**const int** x)
2. {
3. }

In the above syntax, return_type is represented whether the function will return a value or not. The fun_name() function contains a const argument whose value can never be changed once it defines in the program.

Example: Let's consider an example to use the const keyword with function arguments in the C++ programming language.

1. `#include <iostream>`
2. `using namespace std;`
3. `// create an integer Test() function contains an argument num`
4. `int Test (const int num)`
5. `{`
6. `// if we change the value of the const argument, it throws an error.`
7. `// num = num + 10;`
8. `cout << " The value of num: " << num << endl;`
9. `return 0;`
10. `}`
11. `int main ()`
12. `{`
13. `// call function`
14. `Test(5);`
15. `}`

Output

```
The value of num: 5
```

In the above program, the num is a constant argument, and we cannot update the num value. If we update the value of the num variable, it returns the compile-time error.

5. Const Member function of class

1. A const is a constant member function of a class that never changes any class data members, and it also does not call any non-const function.
2. It is also known as the read-only function.
3. We can create a constant member function of a class by adding the const keyword after the name of the member function.

Syntax

1. return_type mem_fun() **const**
2. {
3. }

In the above syntax, mem_fun() is a member function of a class, and the const keyword is used after the name of the member function to make it constant.

Example: Program to use the const keyword with the member function of class

Let's consider an example to define the const keyword with the member function of the class.

1. `class ABC`

```

2. {
3. // define the access specifier
4. public:
5.
6. // declare int variables
7. int A;
8. // declare member function as constant using const keyword
9. void fun () const
10. {
11. A = 0; // it shows compile time error
12. }
13. };
14.
15. int main ()
16. {
17.   ABC obj;
18.   obj.fun();
19.   return 0;
20. }

```

Output

The above code throws a compilation error because the fun() function is a const member function of class ABC, and we are trying to assign a value to its data member 'x' that returns an error.

6. Constant Data Members of class

Data members are like the variable that is declared inside a class, but once the data member is initialized, it never changes, not even in the constructor or destructor. The constant data member is initialized using the const keyword before the data type inside the class. The const data members cannot be assigned the values during its declaration; however, they can assign the constructor values.

Example: Program to use the const keyword with the Data members of the class

```

1. /* create a program to demonstrate the data member using the const keyword in C++. */
2. #include <iostream>
3. using namespace std;
4. // create a class ABC
5. class ABC
6. {
7.
8. public:
9. // use const keyword to declare const data member
10. const int A;
11. // create class constructor
12. ABC ( int y) : A(y)
13. {
14. cout << " The value of y: " << y << endl;
15. }
16. };
17. int main ()
18. {
19. ABC obj( 10); // here 'obj' is the object of class ABC
20. cout << " The value of constant data member 'A' is: " << obj.A << endl;
21. // obj.A = 5; // it shows an error.
22. // cout << obj.A << endl;
23. return 0;
24. }

```

Output

```
The value of y: 10
The value of constant data member 'A' is: 10
```

In the above program, the object obj of the ABC class invokes the ABC constructor to print the value of y, and then it prints the value of the const data member 'A' is 10. But when the 'obj.A' assigns a new value of 'A' data member, it shows a compile-time error because A's value is constant and cannot be changed.

7. Constant Objects

When we create an object using the const keyword, the value of data members can never change till the life of the object in a program. The const objects are also known as the read-only objects.

Syntax

1. **const** class_name obj_name;

The const is a keyword used before the class' object's name in the above syntax to make it a constant object.

Example: Let's create a program to use the constant objects in the C++ programming language.

```
1. #include <iostream>
2. using namespace std;
3. class ABC
4. {
5. public:
6. // define data member
7. int A;
8. // create constructor of the class ABC
9. ABC ()
10. {
11. A = 10; // define a value to A
12. }
13. };
14. int main ()
15. {
16. // declare a constant object
17. const ABC obj;
18. cout << " The value of A: " << obj.A << endl;
19. // obj.A = 20; // It returns a compile time error
20. return 0;
21. }
```

Output

```
The value of A: 10
```

In the above program, we assign the value of A is 10, the compiler prints "The value of A: 10", and when we assign the value of A to 20, the object of the class returns the compile time error in the program.

Program to find the area of a circle using the const keyword

Let's consider an example to print the area of a circle using the const keyword in the C++ programming language.

```
1. #include <iostream>
2. using namespace std;
3. int main ()
4. {
5. // declare variables
6. int r;
7. const double PI = 3.14;
8. double result;
9. cout << " Input the radius of a circle: " << endl;
10. cin >> r;
11. result = PI * r * r;
```

```
12. cout << " The area of a circle is: " << result;
13.}
```

Output

```
Input the radius of a circle:
5
The area of a circle is: 78.5
```

Memset in C++

In the C++ programming language, `memset ()` is a function used to fill memory blocks.

- Initially, it converts the value of '**ch**' to the unsigned character. Here 'ch' refers to the character to be filled with another value passed in the `memset ()` function.
- After then it copies the character 'ch' into each of the first 'n' characters of the object pointed by the `st []`.
- Here the 'n' is referred to as the size of the block, which is mentioned in the `memset ()`, and it must be equal or smaller with the size of the object pointed by `st []`.
- If the value of n is greater than the size of the object pointed by the `st []`, it will generate error hence undefined.
- If sometimes, a case arises where the object is not copyable, then also it will generate an error, and the behavior of the function is the same as in the previous case, i.e., **undefined**.
- In the C++ programming language, the `memset ()` function is present in the `< cstring >` header file; without mentioning this header file, you would not be able to access the use of the `memset ()` function.
- Here, the object which is not copy-able is as follows: **array, C-compatible struct, scalar**, and so on; hence the behavior of `memset ()` function is undefined in this case.
- The only difference between the `memset ()` function and the `memcpy ()` function is that in `memset ()` function not only copies the value but replace it with the other substitute, e.g., if we want to replace each character of a particular string-like, ' cool ', with the alphabet ' f ', then; as a result, and it would be looking at final is; ' ffff '. But in `memcpy ()` function, it only copies the value from one place to another or copies a block of content from a particular place and puts it on another block of content.

Syntax of memset () function:

- void*** `memset (void* st , int ch , size_t n) ;`

Now, let us discuss the terminologies which are used here.

- st []:** `st []` represents the pointer to the memory required to be filled. It is a pointer to the object, where the character ' ch ' has been copied. As per the programmer's choice, they can change the name from `st` to some other, depending on their need and work.
- ch:** ' ch ' here refers to the **character**, which is needed to be filled. It also varies from programmer to programmer choice, and they can rename it accordingly. It is converted to the unsigned char, whenever required and also it is converted into int format, but int acceptable values are only 0 and -1.
- n:** It denotes the **size** of the block, or can say it the numeric value which specifies the number of times the character ' ch ' needed to be copied and converted into the value passed in the `memset ()` function.

Advantages of memset () function of C++

- Increase readability:** The `memset ()` function in C++ is mainly used to convert every character of the whole string into a particular int value which is passed as an input into the `memset ()` function. It is a **one-line code**; hence very short and ultimately increases the readability.
- Reduce line of code:** Instead of using unnecessary use of loops to assign and convert the value of each character present in the string with the int value, which is only passed as an input in this `memset ()` function, and the same task has been performed easily as compared with the lengthy method.
- It is more Faster:** Using the `memset ()` function is faster to convert each character of the given string into the value passed through an input, maybe of int type or any other, depending on the programmer. Its working is very fast rather than applying loops and while statements for performing the same task.
- Useful in getting rid of Misalignment Problem:** The `memset ()` function in C++ helps the programmer to get rid of misalignment problem. Sometimes, the case occurs where you find that you are dealing with the problem of misalignment of data in the processor, which leads to the error in the program. In this case, the `memset ()` and `memcpy ()` functions in C++ are the ultimate solutions of it.

Implementation of memset () in C++

As we have already seen the concept of the `memset()` function of C++ and its working, let us understand it in more detail with the help of an example.

Example 1:

```

1. // Program to implement memset ( ) function in C++
2.
3. #include <iostream>
4. #include <cstring> // header file that contains the memset ( ) function, without it // we cannot able to access the use of memset ( ) function
5. using namespace std ;
6.
7. int main()
8. {
9.     char st [ ] = " JavaTpoint " ;
10.    char ch ;
11.    cout << " Printing the given string : " << st << endl ;
12.    cout << " Enter an alphabet from which you want to replace the each character of the given string : " ;
13.    cin >> ch ;
14.    memset ( st , ch , sizeof ( st ) ) ;
15.    cout << " Printing the string after replacing the each character of the string with the given substitute : " << endl ;
16.
17.    cout << " " << st ;
18.    return 0 ;
19. }

```

The output of the above program:

```
Printing the given string : JavaTpoint
Enter an alphabet from which you want to replace the each charater of the given string : r
Printing the string after replacing the each character of the string with the given substitute :
rrrrrrrrrrrr
```

In this above example, we have seen how each of the characters of the string '**JavaTpoint**' has been converted into the single alphabet passed by a user. Here the space is also considered as a single character.

Example 2:

```

1. // Program to implement memset ( ) function in C++
2.
3. #include < iostream >
4. #include < cstring >           // header file that contains the memset ( ) function, without it //we cannot able to access the use
   of memset ( ) function
5. using namespace std ;
6. int main()
7. {
8.     int arr [100] ;
9.     int n , i , p ;
10.    cout << " Enter size of an integer array: " ;
11.    cin >> n ;
12.    cout << " Enter an array elements: " << endl ;
13.    for ( i = 0 ; i < n ; i++ )
14.    {
15.        cin >> arr [ i ] ;
16.    }
17.    cout << " Printing the given array : " << endl ;
18.    for ( i = 0 ; i < n ; i++ )
19.    {
20.        cout << " " << arr [ i ] << endl ;
21.    }

```

```

22. cout << " Enter an integer value from which you want to replace the each charater of the given string : ";
23. cin >> p ;
24. memset ( arr , p , sizeof( arr ) ) ;
25. cout << " Printing the array after replacing the each value of the array with the given substitute : " << endl ;
26. for ( i = 0 ; i < n ; i++ )
27. {
28.     cout << " " << arr [ i ] << endl ;
29. }
30.
31. return 0 ;
32. }

```

The output of the above program:

```

Enter size of an integer array: 6
Enter an array elements:
14
15
12
10
78
111
Printing the given array :
14
15
12
10
78
111
Enter an integer value from which you want to replace the each charater of the given string : 0
Printing the array after replacing the each value of the array with the given substitute :
0
0
0
0
0
0
0

```

In this above example, we have seen how each of the characters of the string 'JavaTpoints' has been converted into the integer value passed by a user. Here the space is also considered as a single character. Here the integer value only 0 and -1 is considerable, which implies that each character of the string is converted into the 0's and -1's only, instead of using other integer values, will result in a garbage value, which implies that if you have entered 2, then all the characters of the string has been replaced by a garbage value.

Type Casting in C++

This section will discuss the type casting of the variables in the C++ programming language. Type casting refers to the conversion of one data type to another in a program. Typecasting can be done in two ways: automatically by the compiler and manually by the programmer or user. Type Casting is also known as Type Conversion.



For example, suppose the given data is an integer type, and we want to convert it into float type. So, we need to manually cast int data to the float type, and this type of casting is called the Type Casting in C++.

```

1. int num = 5;
2. float x;

```

3. `x = float(num);`
4. `x = 5.0`

2nd example:

1. `float num = 5.25;`
2. `int x;`
3. `x = int(num);`
4. Output: 5

Type Casting is divided into two types: Implicit conversion or Implicit Type Casting and Explicit Type Conversion or Explicit Type Casting.

Implicit Type Casting or Implicit Type Conversion

- It is known as the automatic type casting.
- It automatically converted from one data type to another without any external intervention such as programmer or user. It means the compiler automatically converts one data type to another.
- All data type is automatically upgraded to the largest type without losing any information.
- It can only apply in a program if both variables are compatible with each other.

1. `char` - sort `int` -> `int` -> unsigned `int` -> `long int` -> `float` -> `double` -> `long double`, etc.

Note: Implicit Type Casting should be done from low to higher data types. Otherwise, it affects the fundamental data type, which may lose precision or data, and the compiler might flash a warning to this effect.

Program to use the implicit type casting in C++

Let's create an example to demonstrate the casting of one variable to another using the implicit type casting in C++.

```
1. #include <iostream>
2. using namespace std;
3. int main ()
4. {
5.     short x = 200;
6.     int y;
7.     y = x;
8.     cout << " Implicit Type Casting " << endl;
9.     cout << " The value of x: " << x << endl;
10.    cout << " The value of y: " << y << endl;
11.
12.    int num = 20;
13.    char ch = 'a';
14.    int res = 20 + 'a';
15.    cout << " Type casting char to int data type ('a' to 20): " << res << endl;
16.
17.    float val = num + 'A';
18.    cout << " Type casting from int data to float type: " << val << endl;
19.    return 0;
20. }
```

Output:

```
Implicit Type Casting
The value of x: 200
The value of y: 200
Type casting char to int data type ('a' to 20): 117
Type casting from int data to float type: 85
```

In the above program, we declared a short data type variable x is 200 and an integer variable y. After that, we assign x value to the y, and then the compiler automatically converts short data value x to the y, which returns y is 200.

In the next expressions, we declared an int type variable num is 20, and the character type variable ch is 'a', which is equivalent to an integer value of 97. And then, we add these two variables to perform the implicit conversion, which returns the result of the expression is 117.

Similarly, in the third expression, we add the integer variable num is 20, and the character variable ch is 65, and then assign the result to the float variable val. Thus the result of the expression is automatically converted to the float type by the compiler.

Explicit Type Casting or Explicit Type Conversion

- It is also known as the manual type casting in a program.
- It is manually cast by the programmer or user to change from one data type to another type in a program. It means a user can easily cast one data to another according to the requirement in a program.
- It does not require checking the compatibility of the variables.
- In this casting, we can upgrade or downgrade the data type of one variable to another in a program.
- It uses the cast () operator to change the type of a variable.

Syntax of the explicit type casting

1. (type) expression;

type: It represents the user-defined data that converts the given expression.

expression: It represents the constant value, variable, or an expression whose data type is converted.

For example, we have a floating pointing number is 4.534, and to convert an integer value, the statement as:

1. **int** num;
2. num = (**int**) 4.534; // cast into int data type
3. cout << num;

When the above statements are executed, the floating-point value will be cast into an integer data type using the cast () operator. And the float value is assigned to an integer num that truncates the decimal portion and displays only 4 as the integer value.

Program to demonstrate the use of the explicit type casting in C++

Let's create a simple program to cast one type variable into another type using the explicit type casting in the C++ programming language.

1. **#include <iostream>**
2. **using namespace** std;
3. **int** main ()
4. {
5. // declaration of the variables
6. **int** a, b;
7. **float** res;
8. a = 21;
9. b = 5;
10. cout << " Implicit Type Casting: " << endl;
11. cout << " Result: " << a / b << endl; // it loses some information
- 12.
13. cout << " \n Explicit Type Casting: " << endl;
14. // use cast () operator to convert int data to float
15. res = (**float**) 21 / 5;
16. cout << " The value of float variable (res): " << res << endl;
- 17.
18. **return** 0;
19. }

Output:

```
Implicit Type Casting:
Result: 4

Explicit Type Casting:
The value of float variable (res): 4.2
```


In the above program, we take two integer variables, a and b, whose values are 21 and 2. And then, divide a by b (21/2) that returns a 4 int type value.

In the second expression, we declare a float type variable res that stores the results of a and b without losing any data using the cast operator in the explicit type cast method.

Program to cast double data into int and float type using the cast operator

Let's consider an example to get the area of the rectangle by casting double data into float and int type in C++ programming.

```
1. #include <iostream>
2. using namespace std;
3. int main ()
4. {
5.     // declaration of the variables
6.     double l, b;
7.     int area;
8.
9.     // convert double data type to int type
10.    cout << " The length of the rectangle is: " << endl;
11.    cin >> l;
12.    cout << " The breadth of the rectangle is: " << endl;
13.    cin >> b;
14.    area = (int) l * b; // cast into int type
15.    cout << " The area of the rectangle is: " << area << endl;
16.
17.    float res;
18.    // convert double data type to float type
19.    cout << " \n \n The length of the rectangle is: " << l << endl;
20.    cout << " The breadth of the rectangle is: " << b << endl;
21.    res = (float) l * b; // cast into float type
22.    cout << " The area of the rectangle is: " << res;
23.    return 0;
24. }
```

Output:

```
The length of the rectangle is:
57.3456
The breadth of the rectangle is:
12.9874
The area of the rectangle is: 740

The length of the rectangle is: 57.3456
The breadth of the rectangle is: 12.9874
The area of the rectangle is: 744.77
```

Some different types of the Type Casting

In type cast, there is a cast operator that forces one data type to be converted into another data type according to the program's needs. C++ has four different types of the cast operator:

1. Static_cast
2. dynamic_cast
3. const_cast
4. reinterpret_cast

Static Cast:

The static_cast is a simple compile-time cast that converts or cast one data type to another. It means it does not check the data type at runtime whether the cast performed is valid or not. Thus the programmer or user has the responsibility to ensure that the conversion was safe and valid.

The static_cast is capable enough that can perform all the conversions carried out by the implicit cast. And it also performs the conversions between pointers of classes related to each other (upcast - > from derived to base class or downcast - > from base to derived class).

Syntax of the Static Cast

1. **static_cast** < new_data_type> (expression);

Program to demonstrate the use of the Static Cast

Let's create a simple example to use the static cast of the type casting in C++ programming.

```
1. #include <iostream>
2. using namespace std;
3. int main ()
4. {
5.     // declare a variable
6.     double l;
7.     l = 2.5 * 3.5 * 4.5;
8.     int tot;
9.
10.    cout << " Before using the static cast:" << endl;
11.    cout << " The value of l = " << l << endl;
12.
13.    // use the static_cast to convert the data type
14.    tot = static_cast < int > (l);
15.    cout << " After using the static cast: " << endl;
16.    cout << " The value of tot = " << tot << endl;
17.
18.    return 0;
19.}
```

Output:

```
Before using the static cast:
The value of l = 39.375
After using the static cast:
The value of tot = 39
```

Dynamic Cast

The dynamic_cast is a runtime cast operator used to perform conversion of one type variable to another only on class pointers and references. It means it checks the valid casting of the variables at the run time, and if the casting fails, it returns a NULL value. Dynamic casting is based on RTTI (Runtime Type Identification) mechanism.

Program to demonstrate the use of the Dynamic Cast in C++

Let's create a simple program to perform the dynamic_cast in the C++ programming language.

```
1. #include <iostream>
2. using namespace std;
3.
4. class parent
5. {
6.     public: virtual void print()
7.     {
8.
9.     }
10. };
11. class derived: public parent
12. {
13.
14. };
15.
16. int main ()
17. {
18.     // create an object ptr
```

```

19. parent *ptr = new derived;
20. // use the dynamic cast to convert class data
21. derived* d = dynamic_cast <derived*> (ptr);
22.
23. // check whether the dynamic cast is performed or not
24. if ( d != NULL)
25. {
26.     cout << " Dynamic casting is done successfully";
27. }
28. else
29. {
30.     cout << " Dynamic casting is not done successfully";
31. }
32. }

```

Output:

```
Dynamic casting is done successfully.
```

Reinterpret Cast Type

The reinterpret_cast type casting is used to cast a pointer to any other type of pointer whether the given pointer belongs to each other or not. It means it does not check whether the pointer or the data pointed to by the pointer is the same or not. And it also cast a pointer to an integer type or vice versa.

Syntax of the reinterpret_cast type

1. **reinterpret_cast** <type> expression;

Program to use the Reinterpret Cast in C++

Let's write a program to demonstrate the conversion of a pointer using the reinterpret in C++ language.

```

1. #include <iostream>
2. using namespace std;
3.
4. int main ()
5. {
6.     // declaration of the pointer variables
7.     int *pt = new int (65);
8.
9.     // use reinterpret_cast operator to type cast the pointer variables
10.    char *ch = reinterpret_cast <char *> (pt);
11.
12.    cout << " The value of pt: " << pt << endl;
13.    cout << " The value of ch: " << ch << endl;
14.
15.    // get value of the defined variable using pointer
16.    cout << " The value of *ptr: " << *pt << endl;
17.    cout << " The value of *ch: " << *ch << endl;
18.    return 0;
19.
20. }

```

Output:

```

The value of pt: 0x5cfed0
The value of ch: A
The value of *ptr: 65
The value of *ch: A

```

Const Cast

The `const_cast` is used to change or manipulate the `const` behavior of the source pointer. It means we can perform the `const` in two ways: setting a `const` pointer to a non-`const` pointer or deleting or removing the `const` from a `const` pointer.

Syntax of the Const Cast type

1. `const_cast` <type> exp;

Program to use the Const Cast in C++

Let's write a program to cast a source pointer to a non-cast pointer using the `const_cast` in C++.

```
1. #include <iostream>
2. using namespace std;
3.
4. // define a function
5. int disp(int *pt)
6. {
7.     return (*pt * 10);
8. }
9.
10. int main ()
11. {
12.     // declare a const variable
13.     const int num = 50;
14.     const int *pt = # // get the address of num
15.
16.     // use const_cast to chnage the constness of the source pointer
17.     int *ptr = const_cast <int *> (pt);
18.     cout << " The value of ptr cast: " << disp(ptr);
19.     return 0;
20.
21. }
```

Output:

```
The value of ptr cast: 500
```

Binary Operator Overloading in C++

This section will discuss the Binary Operator Overloading in the C++ programming language. An operator which contains two operands to perform a mathematical operation is called the Binary Operator Overloading. It is a polymorphic compile technique where a single operator can perform various functionalities by taking two operands from the programmer or user. There are multiple binary operators like `+`, `-`, `*`, `/`, etc., that can directly manipulate or overload the object of a class.



For example, suppose we have two numbers, 5 and 6; and overload the binary `(+)` operator. So, the binary `(+)` operator adds the numbers 5 and 6 and returns 11. Furthermore, we can also perform subtraction, multiplication, and division operation to use the binary operator for various calculations.

Syntax of the Binary Operator Overloading

Following is the Binary Operator Overloading syntax in the C++ Programming language.

1. return_type :: operator binary_operator_symbol (arg)
2. {
3. // function definition
4. }

Here,

return_type: It defines the return type of the function.

operator: It is a keyword of the function overloading.

binary_operator_symbol: It represents the binary operator symbol that overloads a function to perform the calculation.

arg: It defines the argument passed to the function.

Steps to Overload the Binary Operator to Get the Sum of Two Complex Numbers

Step 1: Start the program.

Step 2: Declare the class.

Step 3: Declare the variables and their member function.

Step 4: Take two numbers using the user-defined inp()function.

Step 6: Similarly, define the binary (-) operator to subtract two numbers.

Step 7: Call the print() function to display the entered numbers.

Step 8: Declare the class objects x1, y1, sum, and sub.

Step 9: Now call the print() function using the x1 and y1 objects.

Step 10: After that, get the object sum and sub result by adding and subtracting the objects using the '+' and '-' operators.

Step 11: Finally, call the print() and print2() function using the x1, y1, sum, and sub.

Step 12: Display the addition and subtraction of the complex numbers.

Step 13: Stop or terminate the program.

Example 1: Program to perform the addition and subtraction of two complex numbers using the binary (+) and (-) operator

Let's create a program to calculate the addition and subtraction of two complex numbers by overloading the '+' and '-' binary operators in the C++ programming language.

1. /* use binary (+) operator to add two complex numbers. */
2. #include <iostream>
3. using namespace std;
4. class Complex_num
5. {
6. // declare data member or variables
7. int x, y;
8. public:
9. // create a member function to take input
10. void inp()
11. {
12. cout << " Input two complex number: " << endl;
13. cin >> x >> y;
14. }
15. // use binary '+' operator to overload
16. Complex_num operator + (Complex_num obj)
17. {
18. // create an object
19. Complex_num A;

```

20.     // assign values to object
21.     A.x = x + obj.x;
22.     A.y = y + obj.y;
23.     return (A);
24. }
25. // overload the binary (-) operator
26. Complex_num operator - (Complex_num obj)
27. {
28.     // create an object
29.     Complex_num A;
30.     // assign values to object
31.     A.x = x - obj.x;
32.     A.y = y - obj.y;
33.     return (A);
34. }
35. // display the result of addition
36. void print()
37. {
38.     cout << x << " + " << y << "i" << "\n";
39. }
40.
41. // display the result of subtraction
42. void print2()
43. {
44.     cout << x << " - " << y << "i" << "\n";
45. }
46. };
47. int main ()
48. {
49. Complex_num x1, y1, sum, sub; // here we created object of class Addition i.e x1 and y1
50. // accepting the values
51. x1.inp();
52. y1.inp();
53. // add the objects
54. sum = x1 + y1;
55. sub = x1 - y1; // subtract the complex number
56. // display user entered values
57. cout << "\n Entered values are: \n";
58. cout << " \t";
59. x1.print();
60. cout << " \t";
61. y1.print();
62. cout << "\n The addition of two complex (real and imaginary) numbers: ";
63. sum.print(); // call print function to display the result of addition
64. cout << "\n The subtraction of two complex (real and imaginary) numbers: ";
65. sub.print2(); // call print2 function to display the result of subtraction
66. return 0;
67. }

```

Output

```

Input two complex numbers:
5
7
Input two complex numbers:
3
5
Entered values are:
    5 + 7i
    3 + 5i
The addition of two complex (real and imaginary) numbers: 8 + 12i
The subtraction of two complex (real and imaginary) numbers: 2 - 2i

```

In the above program, we take two numbers from the user, and then use the binary operator to overload the '+' and '-' operators to add and subtract two complex numbers in a class.

Example 2: Program to add two numbers using the binary operator overloading

Let's create a program to calculate the sum of two numbers in a class by overloading the binary plus(+) operator in the C++ programming language.

```
1.  /* use binary (+) operator to perform the addition of two numbers. */
2.  #include <iostream>
3.  using namespace std;
4.  class Arith_num
5.  {
6.      // declare data member or variable
7.      int x, y;
8.      public:
9.          // create a member function to take input
10.         void input()
11.         {
12.             cout << " Enter the first number: ";
13.             cin >> x;
14.         }
15.         void input2()
16.         {
17.             cout << " Enter the second number: ";
18.             cin >> y;
19.         }
20.         // overloading the binary '+' operator to add number
21.         Arith_num operator + (Arith_num &ob)
22.         {
23.             // create an object
24.             Arith_num A;
25.             // assign values to object
26.             A.x = x + ob.x;
27.             return (A);
28.         }
29.         // display the result of binary + operator
30.         void print()
31.         {
32.             cout << "The sum of two numbers is: " <<x;
33.         }
34. };
35. int main ()
36. {
37.     Arith_num x1, y1, res; // here we create object of the class Arith_num i.e x1 and y1
38.     // accepting the values
39.     x1.input();
40.     y1.input();
41.     // assign result of x1 and x2 to res
42.     res = x1 + y1;
43.     // call the print() function to display the results
44.     res.print();
45.     return 0;
46. }
```

Output

```
Enter the first number: 5
Enter the second number: 6
The sum of two numbers is: 11
```

In the above program, we take two numbers, 5 and 6, from the user and then overload the binary plus (+) operator to perform the addition that returns the sum of two numbers is 11.

Example 3: Program to perform the arithmetic operation by overloading the multiple binary operators

Let's create a program to overload multiple binary operators in a class to perform the arithmetic operations.

```
1.  /* use binary operator to perform the arithmetic operations in C++. */
2.  #include <iostream>
3.  using namespace std;
4.  class Arith_num
5.  {
6.      // declare data member or variable
7.      int num;
8.      public:
9.          // create a member function to take input
10.         void input()
11.         {
12.             num = 20; //define value to num variable
13.         }
14.         // use binary '+' operator to add number
15.         Arith_num operator + (Arith_num &ob)
16.         {
17.             // create an object
18.             Arith_num A;
19.             // assign values to object
20.             A.num = num + ob.num;
21.             return (A);
22.         }
23.         // overload the binary (-) operator
24.         Arith_num operator - (Arith_num &ob)
25.         {
26.             // create an object
27.             Arith_num A;
28.             // assign values to object
29.             A.num = num - ob.num;
30.             return (A);
31.         }
32.         // overload the binary (*) operator
33.         Arith_num operator * (Arith_num &ob)
34.         {
35.             // create an object
36.             Arith_num A;
37.             // assign values to object
38.             A.num = num * ob.num;
39.             return (A);
40.         }
41.         // overload the binary (/) operator
42.         Arith_num operator / (Arith_num &ob)
43.         {
44.             // create an object
45.             Arith_num A;
46.             // assign values to object
47.             A.num = num / ob.num;
48.             return (A);
49.         }
50.         // display the result of arithmetic operators
51.         void print()
```



```

52.     {
53.         cout << num;
54.     }
55. };
56. int main ()
57. {
58.     Arith_num x1, y1, res; // here we created object of class Addition i.e x1 and y1
59.     // accepting the values
60.     x1.input();
61.     y1.input();
62.     // assign result of x1 and x2 to res
63.     res = x1 + y1;
64.     cout << " Addition : ";
65.     res.print();
66.     // assign the results of subtraction to res
67.     res = x1 - y1; // subtract the complex number
68.     cout << " \n \n Subtraction : ";
69.     res.print();
70.     // assign the multiplication result to res
71.     res = x1 * y1;
72.     cout << " \n \n Multiplication : ";
73.     res.print();
74.     // assign the division results to res
75.     res = x1 / y1;
76.     cout << " \n \n Division : ";
77.     res.print();
78.     return 0;
79. }

```

Output

```

Addition : 40
Subtraction : 0
Multiplication : 400
Division : 1

```

In the above program, we declare the value of variable num is 20 and then overload the binary plus (+), minus (-), multiply (*), and division (/) operator to perform the various arithmetic operations in the Arith_num class.

Binary Search in C++

We will discuss the binary search in the C++ programming language. Binary search is a mechanism used to find the given elements from the sorted array by continuously halving the array and then searching specified elements from a half array. And the process goes on till the match is found. It works only the sorted data structures. The time complexity of the binary search algorithm is $O(\log n)$.



Binary Search in C++

Note: To perform a binary search technique in C++, a programmer or user should ensure the given array must be sorted either be in ascending or descending order.

Algorithm of the binary search in C++

Following is the algorithm to perform the binary search in C++

```

1. beg = 0;
2. end = size - 1;
3. while ( beg <= end)
4. {
5. // calculate mid value
6. mid = (beg + end) / 2;
7. /* if the specified element is found at mid index, terminate the process and return the index. */
8. // Check middle element is equal to the defined element.
9. If (aarr[mid] == num)
10. {
11. return mid + 1;
12. }
13. else if (arr[mid] > num)
14. {
15. End = mid - 1;
16. }
17. Else if (arr [mid] < num)
18. {
19. Beg = mid + 1;
20. }
21. }
22. // If the element does not exist in the array, return -1.
23. Return -1;

```

Steps to perform the binary search in C++

Step 1: Declare the variables and input all elements of an array in sorted order (ascending or descending).

Step 2: Divide the lists of array elements into halves.

Step 3: Now compare the target elements with the middle element of the array. And if the value of the target element is matched with the middle element, return the middle element's position and end the search process.

Step 4: If the target element is less than the middle element, we search the elements into the lower half of an array.

Step 5: If the target element is larger than the middle element, we need to search the element into the greater half of the array.

Step 6: We will continuously repeat steps 4, 5, and 6 till the specified element is not found in the sorted array.

Example 1: Program to find the specified number from the sorted array using the binary search

Let's write a program to find the specified number from a sorted array using the binary search in the C++ programming language.

```

1. #include <iostream>
2. #include <conio.h>
3. using namespace std;
4. int main ()
5. {
6. // declaration of the variables and array
7. int arr[100], st, mid, end, i, num, tgt;
8.
9. cout << " Define the size of the array: " << endl;
10. cin >> num; // get size
11.
12. // enter only sorted array
13. cout << " Enter the values in sorted array either ascending or descending order: " << endl;
14. // use for loop to iterate values
15. for (i = 0; i < num; i++)
16. {
17. cout << " arr [" << i << "] = ";

```

```

18.     cin >> arr[i];
19. }
20.
21. // initialize the starting and ending variable's values
22. st = 0;
23. end = num - 1; // size of array (num) - 1
24.
25. // define the item or value to be search
26. cout << " Define a value to be searched from sorted array: " << endl;
27. cin >> tgt;
28.
29. // use while loop to check 'st', should be less than equal to 'end'.
30. while ( st <= end)
31. {
32.     // get middle value by splitting into half
33.     mid = ( st + end ) / 2;
34.     /* if we get the target value at mid index, print the position and exit from the program. */
35.     if (arr[mid] == tgt)
36.     {
37.         cout << " Element is found at index " << (mid + 1);
38.         exit (0); // use for exit program the program
39.     }
40.     // check the value of target element is greater than the mid element' value
41.     else if ( tgt > arr[mid])
42.     {
43.         st = mid + 1; // set the new value for st variable
44.     }
45.
46.     // check the value of target element is less than the mid element' value
47.     else if ( tgt < arr[mid])
48.     {
49.         end = mid - 1; // set the new value for end variable
50.     }
51. }
52. cout << " Number is not found. " << endl;
53. return 0;
54. }

```

Output

```

Define the size of the array:
10
Enter the values in sorted array either ascending or descending order:
Arr [0]  = 12
Arr [1]  = 24
Arr [2]  = 36
Arr [3]  = 48
Arr [4]  = 50
Arr [5]  = 54
Arr [6]  = 58
Arr [7]  = 60
Arr [8]  = 72
Arr [9]  = 84
Define a value to be searched from sorted array:
50
Element is found at index 5

```

Example 2: Program to perform the binary search using the user-defined function

1. /* program to find the specified element from the sorted array using the binary search in C++. */
2. #include <iostream>
3. using namespace std;
4. /* create user-defined function and pass different parameters:
5. arr[] - it represents the sorted array;
6. num variable represents the size of an array;

```

7.  tgt variable represents the target or specified variable to be searched in the sorted array. */
8.  int bin_search (int arr[], int num, int tgt)
9.  {
10.     int beg = 0, end = num - 1;
11.     // use loop to check all sorted elements
12.     while (beg <= end)
13.     {
14.         /* get the mid value of sorted array and then compares with target element. */
15.         int mid = (beg + end) / 2;
16.         if (tgt == arr[mid])
17.         {
18.             return mid; // when mid is equal to tgt value
19.         }
20.         // check tgt is less than mid value, discard left element
21.         else if (tgt < arr[mid])
22.         {
23.             end = mid - 1;
24.         }
25.         // if the target is greater than the mid value, discard all elements
26.         else {
27.             beg = mid + 1;
28.         }
29.     }
30.     // return -1 when target is not exists in the array
31.     return -1;
32. }
33. int main ()
34. {
35.     // declaration of the arrays
36.     int arr[] = { 5, 10, 15, 20, 25, 30, 37, 40};
37.     int tgt = 25; // specified the target element
38.     int num = sizeof (arr) / sizeof (arr[0]);
39.     // declare pos variable to get the position of the specified element
40.     int pos = bin_search (arr, num, tgt);
41.     if (pos != 1)
42.     {
43.         cout << " Element is found at position " << (pos + 1)<< endl;
44.     }
45.     else
46.     {
47.         cout << " Element is not found in the array" << endl;
48.     }
49.     return 0;
50. }

```

Output

```
Element is found at position 5
```

In the above program, we declared an array `arr[] = {5, 10, 15, 20, 25, 30, 35, 40}`; and then we specified number '25' to which search from the sorted array using the binary search method. Therefore, we create a user-defined function `bin_search()` that searches the given number and returns the statement "Element is found at position 5". If the number is not defined in the array, the `bin_search()` function shows " Element is not found in the array."

Example 3: Program to find the specified element using the recursion function

Let's create an example to check whether the specified element is found in the sorted array using the binary search inside the recursion function.

```

1.  /* find the specified number using the binary search technique inside the recursion method. */

```

```

2. #include <iostream>
3. using namespace std;
4. // define a function
5. int binary_search (int [], int, int, int);
6. int main ()
7. {
8.     // declaration of the variables
9.     int i, arr[100], tgt, num, ind, st, end;
10.    cout << " Define the size of an array: ";
11.    cin >> num;
12.    cout << " Enter " << num << " elements in ascending order: " << endl;
13.    // use for loop to iterate the number
14.    for ( i = 0; i < num; i++)
15.    {
16.        cout << " arr [" << i << "] = ";
17.        cin >> arr[i];
18.    }
19.    // define the element to be search
20.    cout << " \n Enter an element to be searched in ascending array: ";
21.    cin >> tgt;
22.    // ind define the index number
23.    ind = binary_search (arr, 0, num - 1, tgt);
24.    // check for existemce of the specified element
25.    if (ind == 0)
26.        cout << tgt << " is not available in the array-list";
27.    else
28.        cout << tgt << " is available at position " << ind << endl;
29.    return 0;
30. }
31. // function defnition
32. int binary_search (int arr[], int st, int end, int tgt)
33. {
34.     int mid;
35.     // check st is greater than end
36.     if (st > end)
37.     {
38.         return 0;
39.     }
40.     mid = (st + end) / 2; // get middle value of the sorted array
41.
42.     // check middle value is equal to target number
43.     if (arr[mid] == tgt)
44.     {
45.         return (mid + 1);
46.     }
47.     // check mid is greater than target number
48.     else if (arr[mid] > tgt)
49.     {
50.         binary_search (arr, st, mid - 1, tgt);
51.     }
52.     // check mid is less than target number
53.     else if (arr [mid] < tgt)
54.     {
55.         binary_search (arr, mid + 1, end, tgt);
56.     }
57. }

```

Output

```
Define the size of an array: 10
Arr [0] = 2
Arr [1] = 4
Arr [2] = 5
Arr [3] = 8
Arr [4] = 12
Arr [5] = 13
Arr [6] = 27
Arr [7] = 36
Arr [8] = 49
Arr [9] = 51

Enter an element to be searched in ascending array: 12
12 is available at position 6.
```

In the above program, we input all elements of an array in ascending order and then define a number as the target element is '12', which is searched from a sorted array using the binary search method. So, we create a user-defined function `binary_search()` that searches the defined element's position from an array and returns the particular element available at this position. And if the element is not available in the sorted array, it returns 0.