C++

Digital Notes

Good Things, Take Times

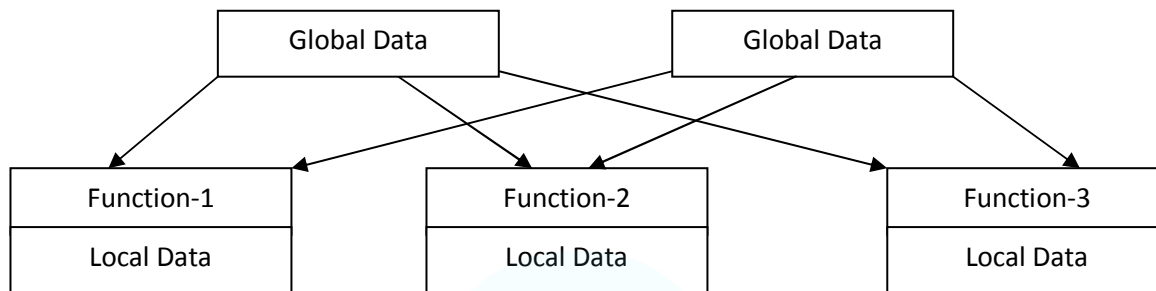# CONTENTS
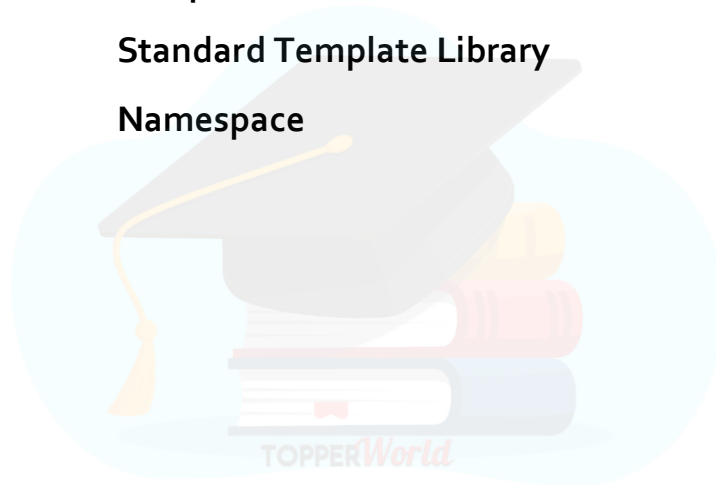
Procedure/ structure oriented Programming

- Conventional programming, using high level languages such as COBOL, FORTRAN and C, is commonly known as procedure-oriented programming (POP).
- In the procedure-oriented approach, the problem is viewed as a sequence of things to be done such as reading, calculating and printing. A number of functions are written to accomplish these tasks.
- The primary focus is on functions.

```
      ┌──────────────┐        ┌──────────────┐
      │ Global Data  │        │ Global Data  │
      └──────────────┘        └──────────────┘
```

Function-1		Function-2		Function-3
Local Data		Local Data		Local Data

Object Oriented Programming

- Emphasis is on data rather than procedure.
- Programs are divided into what are known as objects.
- Data is hidden and cannot be accessed by external functions.
- Objects may communicate with each other through functions.
- New data and functions can be easily added whenever necessary.

Object A
Data
Functions

Object B
Data
Functions

Object C
Data
Functions
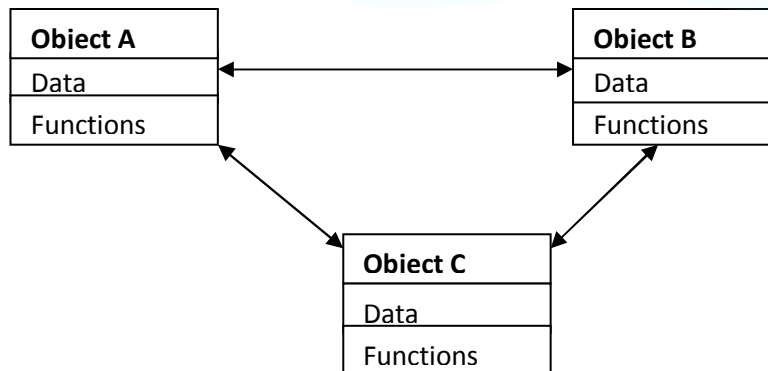
Basic Concepts of Object-Oriented Programming

Objects
Objects are the basic runtime entities in an object oriented system. They may represent a person, a place, a bank account, a table of data or any item that the program has to handle.

Class
Object contains data, and code to manipulate that data. The entire set of data and code of an object can be made a user-defined data type with the help of a class.

Data Encapsulation
- The wrapping up of data and functions into a single unit is known as encapsulation.
- The data is not accessible to the outside world, only those function which are wrapped in the can access it.
- These functions provide the interface between the object's data and the program.
- This insulation of the data from direct access by the program is called **data hiding** or **information hiding**.
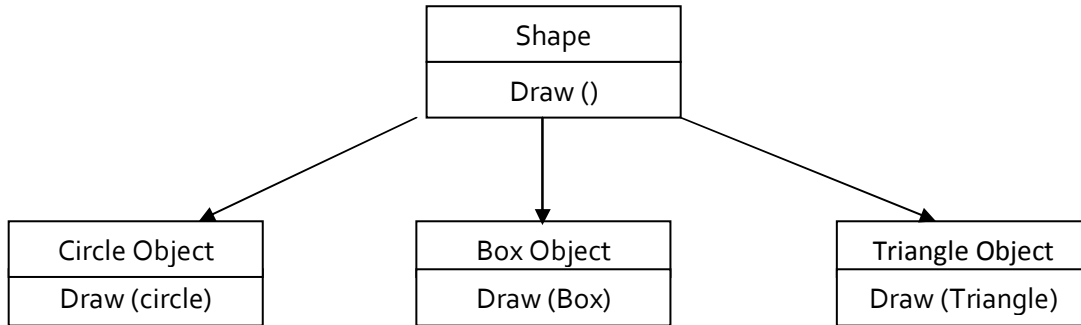
Data Abstraction
- Abstraction refers to the act of representing essential features without including the background details or explanations.
- Since classes use the concept of data abstraction, they are known as **Abstract Data Types (ADT)**.

Inheritance
- **Inheritance** is the process by which objects of one class acquire the properties of objects of another class.
- In OOP, the concept of inheritance provides the idea of reusability. This means we can add additional features to an existing class without modifying it.

Polymorphism
- **Polymorphism,** a Greek term means to ability to take more than one form.
- An operation may exhibits different behaviors in different instances. The behavior depends upon the type of data used in the operation.
- For example consider the operation of addition for two numbers; the operation will generate a sum. If the operands are string then the operation would produce a third string by concatenation.
- The process of making an operator to exhibit different behavior in different instances is known operator overloading.

```
                    ┌──────────────────┐
                    │     Shape        │
                    ├──────────────────┤
                    │    Draw ()       │
                    └──────────────────┘
```

┌───────────────────┐   ┌───────────────────┐   ┌───────────────────┐
│ Circle Object   │   │   Box Object   │   │ Triangle Object │
├───────────────────┤   ├───────────────────┤   ├───────────────────┤
│ Draw (circle)   │   │   Draw (Box)   │   │ Draw (Triangle) │
└───────────────────┘   └───────────────────┘   └───────────────────┘

Output and Input Statement in C++

An Output statement is used to print the output on computer screen. cout is an output statement.

cout<<**"**Srinix College of Engineering**"**; prints **Srinix College of Engineering** on computer screen.

cout<<**"**x**"**; print **x** on computer screen.

cout<<x; prints **value of x** on computer screen.

cout<<**"\n"**; takes the cursor to a newline.

cout<< **endl**; takes the cursor to a newline. We can use **endl** (a manipulator) instead of **\n**.

<< (two "less than" signs) is called insertion operator.

An Input statement is used to take input from the keyboard. cin is an input statement.

cin>>x; takes the value of x from keyboard.

cin>>x>>y; takes value of x and y from the keyboard.

Program 1.1 WAP to accept an integer from the keyboard and print the number when it is multiplied by 2.

Solution:
```
#include <iostream.h>
void main ()
{
 int x;
 cout << "Please enter an integer value: ";
 cin >>x;
 cout <<endl<< "Value you entered is " <<x;
 cout << " and its double is " <<x*2 << ".\n";
}
```
Output:

Please enter an integer value:

5
Value you entered is 5 and its double is 10.

1. Arithmetic Operators (+, -, *, /, %)

The five arithmetical operations supported by the C language are:

Addition (+)
Subtraction (-)
Multiplication (*)
Division (/)
Modulo (%)

Operations of addition, subtraction, multiplication and division literally correspond with their respective mathematical operators.

Division Rule

Integer/integer=integer
Integer/float=float
Float/integer=float
Float /float=float

Modular division

a>=b	a%b =remainder when a is divided by b
a<b	a

Program 1.2   **Write a program to demonstrate arithmetic operation.**

Solution:

```
#include <iostream.h>
void main ()
{
        int a, b, p, q, r, s;
        a = 10;
        b=4;
        p= a/b;
        q= a*b;
        r= a%b;
        s= b%a;
        cout<<p<<q<<r<<s;
}
```

Output:

2 40 2 4

2. Assignment Operator (=)

The assignment operator assigns a value to a variable.

a = 5;

This statement assigns the integer value 5 to the variable a. The part at the left of the assignment operator (=) is known as the lvalue (left value) and the right one as the rvalue (right value). The lvalue has to be a variable whereas the rvalue can be either a constant, a variable, the result of an operation or any combination of these. The most important rule when assigning is the right-to-left rule: The assignment operation always takes place from right to left,

and never the other way:

a = b;

This statement assigns to variable a (the lvalue) the value contained in variable b (the rvalue). The value that was stored until this moment in a is not considered at all in this operation, and in fact that value is lost.

Shorthand assignment (+=, -=, *=, /=, %=, >>=, <<=, &=, ^=, |=)

When we want to modify the value of a variable by performing an operation on the value currently stored in that variable we can use compound assignment operators:

value += increase; is equivalent to value = value + increase;

a -= 5; is equivalent to a = a - 5;

a /= b; is equivalent to a = a / b;

price *= units + 1; is equivalent to price = price * (units + 1); and the same for all other operators.

3. Relational and equality operators (==, !=, >, <, >=, <=)

In order to evaluate a comparison between two expressions we can use the relational and equality operators. The result of a relational operation is a Boolean value that can only be true or false, according to its Boolean result. We may want to compare two expressions, for example, to know if they are equal or if one is greater than the other is. Here is a list of the relational and equality operators that can be used in C++:

Here there are some examples:

(7 == 5) // evaluates to false.

(5 > 4) // evaluates to true.

(3 != 2) // evaluates to true.

(6 >= 6) // evaluates to true.

(5 < 5) // evaluates to false.

Of course, instead of using only numeric constants, we can use any valid expression, including variables. Suppose that a=2, b=3 and c=6,

(a == 5) // evaluates to false since a is not equal to 5.

(a*b >= c) // evaluates to true since (2*3 >= 6) is true.

(b+4 > a*c) // evaluates to false since (3+4 > 2*6) is false.

((b=2) == a) // evaluates to true.

Important Tips!

Be careful! The operator = (one equal sign) is not the same as the operator == (two equal signs), the first one is an assignment operator (assigns the value at its right to the variable at its left) and the other one

(==) is the equality operator that compares whether both expressions in the two sides of it are equal to each other. Thus, in the last expression ((b=2) == a), we first assigned the value 2 to b and then we compared it to a, that also stores the value 2, so the result of the operation is true.

4. Logical operators (!, &&, ||)

The Operator! is the C++ operator to perform the Boolean operation NOT, it has only one operand, located at its right, and the only thing that it does is to inverse the value of it, producing false if its operand is true and true if its operand is false. Basically, it returns the opposite Boolean value of evaluating its operand.

For example: !(5 == 5) // evaluates to false because the expression at its right (5 == 5) is true. !(6 <= 4) // evaluates to true because (6 <= 4) would be false. !true // evaluates to false. !false // evaluates to true. The logical operators && and || are used when evaluating two expressions to obtain a single relational result. The operator && corresponds with Boolean logical operation AND. This operation results true if both its two operands are true, and false otherwise. The following panel shows the result of operator && evaluating the expression a && b:

a	b	a && b
True	True	True
True	False	False
False	True	False
False	False	False

The operator || corresponds with Boolean logical operation OR. This operation results true if either one of its two operands is true, thus being false only when both operands are false themselves. Here are the possible results of a || b:

a	b	a || b
True	True	True
True	False	True
False	True	True
False	False	False

For example:

((5 == 5) && (3 > 6)) // evaluates to false (true && false).

((5 == 5) || (3 > 6)) // evaluates to true (true || false).

5. Increment and Decrement Operator (++, --)

The increment operator (++) and the decrement operator (--) increase or reduce by one the value stored in a variable. They are equivalent to +=1 and to - =1, respectively. Thus:

c++;

c+=1;

c=c+1;

are all equivalent in its functionality: the three of them increase by one the value of c.
A characteristic of this operator is that it can be used both as a prefix and as a suffix. That means that it can be written either before the variable identifier (++a) or after it (a++). Although in simple expressions like a++ or ++a both have exactly the same meaning, in other expressions in which the result of the increase or decrease operation is evaluated as a value in an outer expression they may have an important difference in their meaning: In the case that the increase operator is used as a prefix (++a) the value is increased before the result of the expression is evaluated and therefore the increased value is considered in the outer expression; in case that it is used as a suffix (a++) the value stored in a is increased after being evaluated and therefore the value stored before the increase operation is evaluated in the outer expression.

Pre	First increment/decrement then assignment.
Post	First assignment then increment/decrement.

Notice the difference:

Example 1 **Find the value of A and B.**

B=3;
A=++B;
Ans: A contains 4, B contains 4

Example 2 **Find the value of A and B.**

B=3;
A=B++;
Ans: A contains 3, B contains 4

In Example 1, B is increased before its value is assigned to A. While in Example 2, the value of B is assigned to A and then B is increased.

6. Conditional operator (? :)

The conditional operator evaluates an expression returning a value if that expression is true and a different one if the expression is evaluated as false. Its format is: condition ? result1 : result2. If condition is true the expression will return result1, if it is not it will return result2.

7==5 ? 4 : 3 // returns 3, since 7 is not equal to 5.
7==5+2 ? 4 : 3 // returns 4, since 7 is equal to 5+2.
5>3 ? a : b // returns the value of a, since 5 is greater than 3.
a>b ? a : b // returns whichever is greater, a or b.

Program 1.3 **Write a program to find the greatest of two numbers.**

Solution:

#include <iostream.h>

```
void main ()
{
        int a,b,c;
        a=2;
        b=7;
        c = (a>b) ? a : b;
        cout<<c;
}
```

Output: 7

In this example a was 2 and b was 7, so the expression being evaluated (a>b) was not true, thus the first value specified after the question mark was discarded in favor of the second value (the one after the colon) which was b, with a value of 7.

7. Comma operator (,)
The comma operator (,) is used to separate two or more expressions that are included where only one expression is expected. When the set of expressions has to be evaluated for a value, only the rightmost expression is considered.
For example, the following code:
a = (b=3, b+2); would first assign the value 3 to b, and then assign b+2 to variable a. So, at the end, variable a would contain the value 5 while variable b would contain value 3.

8. Explicit type casting operator
Type casting operators allow you to convert a datum of a given type to another. There are several ways to do this in C++. The simplest one, which has been inherited from the C language, is to precede the expression to be converted by the new type enclosed between parentheses (()):
int i;
float f = 3.14;
i = (int) f;
The previous code converts the float number 3.14 to an integer value (3), the remainder is lost. Here, the typecasting operator was (int). Another way to do the same thing in C++ is using the functional notation: preceding the expression to be converted by the type and enclosing the expression between parentheses:
i = int (f);
Both ways of type casting are valid in C++.

9. sizeof ()
This operator accepts one parameter, which can be either a type or a variable itself and returns the size in bytes of that type or object:
a = sizeof (char);
This will assign the value 1 to a because char takes 1 byte of memory. The value returned by sizeof is a constant, so it is always determined before program execution.

Control Statements

A programming language uses control statements to cause the flow of execution to advance and branch based on changes to the state of a program. C++ program control statements can be put into the following categories: **selection, iteration, and jump.**
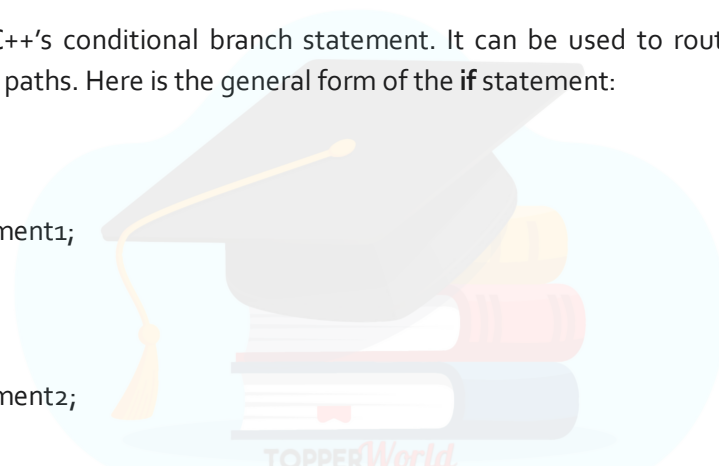
- Selection statements allows program to choose different paths of execution based upon the outcome of an expression or the state of a variable.
- Iteration statements enable program execution to repeat one or more statements (that is, iteration statements form loops).
- Jump statements allows program to execute in a nonlinear fashion. All of Java's control statements are examined here.

Selection Statements

1. if Statement

The **if** statement is C++'s conditional branch statement. It can be used to route program execution through two different paths. Here is the general form of the **if** statement:

```
if (condition)
{
        statement1;
}
else
{
        statement2;
}
```

Here, each statement may be a single statement or a compound statement enclosed in curly braces (that is, a block). The condition is any expression that returns a **boolean** value. The **else** clause is optional. The **if statement works** like this:

If the condition is true, then statement1 is executed. Otherwise, statement2 (if it exists) is executed.

Program 1.4   **WAP to check whether a person is old or young. A person will be said old if his age is above 35 and young if his age is below 35.**

Solution:

```
#include<iostream.h>
void main()
{
        int age;
        cout<<"Enter Age";
```

```
        cin>>age;
        if(age>=35)
        cout<<"Old";
        else
        cout<<"Young";
}
```

Output:
Enter age 39
Old

2. Nested ifs

A nested if is an if statement that is the target of another if or else. Nested ifs are very common in programming. When you nest ifs, the main thing to remember is that an else statement always refers to the nearest if statement that is within the same block as the else and that is not already associated with an else.
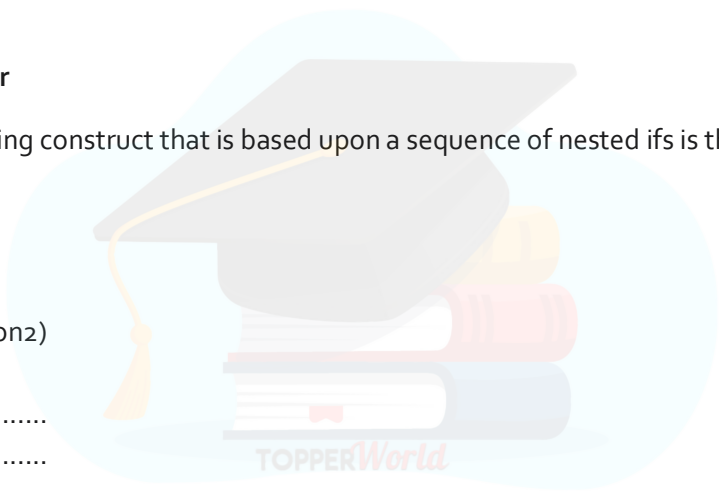
3. The if-else-if Ladder

A common programming construct that is based upon a sequence of nested ifs is the if-else-if ladder. It looks like this:

```
        if (condition1)
        statement1;
        else if (condition2)
        statement2;
        ...........................
        ...........................
        else
        statement3;
```

The if statements are executed from the top to down. As soon as one of the conditions controlling the if is true, the statement associated with that if is executed, and the rest of the ladder is bypassed. If none of the conditions is true, then the final else statement will be executed.

Program 1.5   **Write a program to calculate division obtained by a student with his percentage mark given.**

Solution:
```
#include<iostream.h>
void main()
{
        int mark;
```

```cpp
cout<<"Enter mark";
cin>>mark;

if(mark>=60)
cout<<"1st Division";

else if(mark>=50 && mark<60)
cout<<"2nd Division";

else if(mark>=40 && mark<50)
cout<<"3rd  Division";

else
cout<<"Fail";
}
```

Output:
Enter mark 87
1st Division

Switch Statements

The **switch** statement is C++'s multi way branch statement. It provides an easy way to dispatch execution to different parts of your code based on the value of an expression. As such, it often provides a better alternative than a large series of **if-else-if** statements. Here is the general form of a **switch** statement:

```cpp
switch (expression)
{
        case value1:
        // statement sequence
        break;
        case value2:
        // statement sequence
        break;
        ...
        case valueN:
        // statement sequence
        break;
        default:
        // default statement sequence
```

```
        }
```

Program 1.6 **Write a program to illustrate the use of switch case statements.**

Solution:
```cpp
#include<iostream.h>
void main()
{
        int i
        for(int i=0; i<6; i++)
        switch(i)
        {
        case 0:
        cout<<"i is zero.";
        break;
        case 1:
        cout<<"i is one.";
        break;
        case 2:
        cout<<"i is two.";
        break;
        case 3:
        cout<<"i is three.";
        break;
        default:
        cout<<"i is greater than 3.";
        }
}
```

Output:
i is zero.
i is one.
i is two.
i is three.
i is greater than 3.
i is greater than 3.

Iteration Statements

C++'s iteration statements are for, while, and do-while. These statements create what we commonly call loops. As you probably know, a loop repeatedly executes the same set of instructions until a termination condition is met.

1. while loop

The while loop is Java's most fundamental looping statement. It repeats a statement or block while its controlling expression is true. Here is its general form:

```
initialization
while (condition)
{
// body of loop
Increment/ decrement
}
```

The condition can be any Boolean expression. The body of the loop will be executed as long as the conditional expression is true. When condition becomes false, control passes to the next line of code immediately following the loop. The curly braces are unnecessary if only a single statement is being repeated.

Program 1.7   **Write a program to print the message "C++ is good" 10 times using while loop.**

Solution:
```
#include<iostream.h>
void main()
{
        int n = 0;
        while (n <10)
        {
        cout<<"C++ is good";
        n++;
        }
}
```

2. do while Loop

If the conditional expression controlling a **while** loop is initially false, then the body of the loop will not be executed at all. However, sometimes it is desirable to execute the body of a **while** loop at least once, even if the conditional expression is false to begin with. The **do-while** loop always executes its body at least once, because its conditional expression is at the bottom of the loop. Its general form is:
```
initialization
do
{
// body of loop
```

Increment/ decrement
} while (condition);

Write a program to print the message "C++ is good" 10 times using do while loop.

Solution:
```
#include<iostream.h>
void main()
{
        int n = 0;
        do
        {
        cout<<"C++  is good";
        n++;
        } while(n<9);
}
```

3. for loop
The general form of the **for** statement is:
```
for(initialization; condition; iteration)
 {
// body
}
```

The **for** loop operates as follows:
- When the loop first starts, the initialization portion of the loop is executed. Generally, this is an expression that sets the value of the loop control variable, which acts as a counter that controls the loop. It is important to understand that the initialization expression is only executed once.
- Next, condition is evaluated. This must be a Boolean expression. It usually tests the loop control variable against a target value. If this expression is true, then the body of the loop is executed. If it is false, the loop terminates.
- Next, the iteration portion of the loop is executed. This is usually an expression that increments or decrements the loop control variable. The loop then iterates, first evaluating the conditional expression, then executing the body of the loop, and then executing the iteration expression with each pass. This process repeats until the controlling expression is false.

Program 1.9 **Write a program to print the message "C++is good" 10 times using for loop.**

Solution:
```
#include<iostream.h>
void main()
{
        int n ;
```

```
        for(n=0; n<10; n++)
        {
        cout<<"C++ is good";
        }
}
```

Jump Statements

C++ offers following jump statements:

- **break statement:** A break statement takes control out of the loop.
- **continue statement:** A continue statement takes control to the beginning of the loop.
- **goto statement:** A goto Statement take control to a desired line of a program.

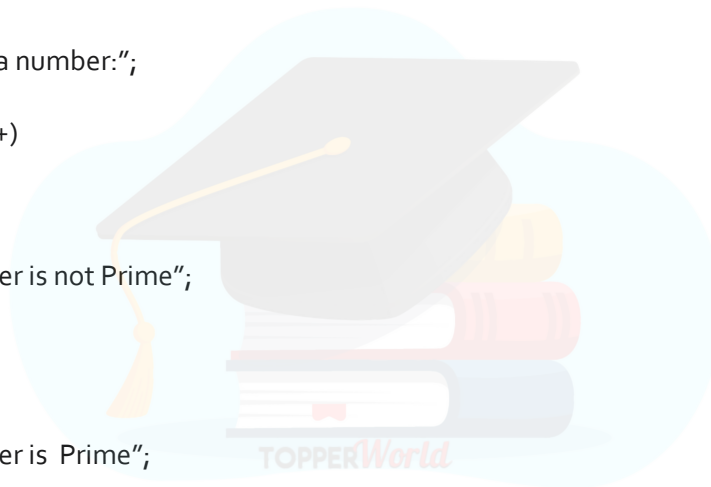**Program 1.10** Write a program to check whether a number is prime or not.

Solution:

```
#include<iostream.h>
void main()
{
        int n, i;
        cout<<"Enter a number:";
        cin>>n;
        for(i=2; i<n; i++)
        {
        if(n%i==0)
        {
        cout<<"Number is not Prime";
        break;
        }
        }
        if(i==n)
        cout<<"Number is  Prime";
}
```

Output:
Enter a number 13
Number is Prime

Program 1.11 The marks obtained by a student in 5 different subjects are input through the keyboard. The student gets a division as per the following rules:
Percentage above or equal to 60 - First division
Percentage between 50 and 59 - Second division
Percentage between 40 and 49 - Third division
Percentage less than 40 – Fail. Write a program to calculate the division obtained by the student.

Solution:
```
#include<iostream.h>
```

```cpp
void main( )
{
        int m1, m2, m3, m4, m5, per ;
        cout<< "Enter marks in five subjects " ;
        cin>>m1>>m2>>m3>>m4>>m5 ;
        per = ( m1 + m2 + m3 + m4 + m5 ) / 5 ;
        if ( per >= 60 )
        {
        cout<< "First division" ;
        }
        else if ( ( per >= 50 ) && ( per < 60 ) )
        {
        cout<< "Second division" ;
        }
        else if ( ( per >= 40 ) && ( per < 50 ) )
        {
        cout<<"Third division" ;
        }
        else
        {
        cout<< "Fail" ;
        }
}
```

Output:
Enter marks in 5 subjects 65 65 65 60 70
First division

Program 1.12   Write a program to find the roots of a quadratic equation $ax^2+bx+c=0$, where the values of coefficient a, b and c are given from the keyboard.

Solution:
```cpp
#include<iostream.h>
#include<math.h>
void main()
{
        int a, b, c, d;
        float r1, r2, x, y;
        cout<<"Enter the coefficients";
        cin>>a>>b>>c;
        d=(b*b) - (4*a*c);
        if(d>0)
```

```
        {
        r1=(-b+pow(d,0.5))/2*a;
        r2=(-b- pow(d,0.5))/2*a;
        cout<<"Roots are real and the roots are "<<r1<<" and "<<r2;
        }
        else if(d==0)
        {
        r1=-b/2*a;
        cout<< "Roots are equal and the root is  "<<r1;
        }
        else
        {
        x=-b/2*a;
        y =pow(-d,0.5)/2*a;
        cout<<"Roots are imaginary and the roots are"<<endl;
        cout<<x<<"+i "<<y<<endl;
        cout<<x<<"-i "<< y;
        }
}
```

Output:
Enter the coefficients 1 -4 4
Roots are equal and the root is 2

Program 1.13   If the three sides of a triangle are entered through the keyboard, write a program to check whether the triangle is isosceles, equilateral, scalene or right angled triangle.

Solution:
```
#include<iostream.h>
void main()
{
        int a, b, c;
        cout<<"Enter 3 sides";
        cin>>a>>b>>c;
        if(a+b>c && b+c>a && c+a>b)
        {
                if((a*a+b*b)==c*c || (b*b+c*c)==a*a || (c*c+a*a)==b*b)
                {
                cout<<"Right Angled";
                }
                else if( (a==b &&b!=c) || (b==c&&c!=a) || (c==a&&a!=b))
                {
```

```cpp
                        cout<<"Isosceles";
                        }
                        else if(a==b && b==c)
                        {
                        cout<<"Equilateral";
                        }
                        else
                        {
                        cout<<"Any Valid Triangle";
                        }
                }
                else
                {
                        cout<<"Cannot form a valid triangle";
                }
}
```

Program 1.14   **Write a program to print all the even integers from 1 to 100.**

Solution:
```cpp
#include<iostream.h>
void main()
{
        int i;
        for(i=0; i<100 ; i+=2)
        {
        cout<< i<<"\n";
        }
}
```

Program 1.15   **Compute 1+2+3+4+.....................+n for a given value of n.**

Solution:
```cpp
#include<iostream.h>
void main()
{
        int i, n, s=0;
        cout<<"Enter the value of n";
        cin>>n;
        for(i=1; i<=n; i++)
        {
```

```
        s=s+i;
        }
        cout<< s;
}
```

Program 1.16 Write a program to compute factorial of a number.

Solution:
```
#include<iostream.h>
void main()
{
        int i, n, f=1;
        cout<<"Enter the value of n";
        cin>>n;
        for(i=1; i<=n; i++)
        {
        f=f*i;
        }
        cout<<f;
}
```

Program 1.17 Write a program to find sum of digits of a number.

Solution:
```
#include<iostream.h>
void main()
{
        int a, n, s=0;
        cout<<"Enter a number";
        cin>>n;
        while(n!=0)
        {
        a=n%10;
        s=s+a;
        n=n/10;
        }
        cout<<s;
}
```

Program 1.18 Write a program to reverse a number.

Solution:

```
#include<iostream.h>
void main()
{
        int a, n, rn=0;
        cout<<"Enter a number";
        cin>>n;
        while(n!=0)
        {
        a=n%10;
        rn=rn*10+a;
        n=n/10;
        }
        cout<<rn;
}
```

Example 1.19 Write a program to check whether a no is palindrome number or not. (A number is said to be palindrome number if it is equal its reverse

Solution:
```
#include<iostream.h>
void main()
{
        int a, n, rn=0, b;
        cout<<"Enter a number";
        cin>>n;
        b=n;
        while(n!=0)
        {
        a=n%10;
        rn=rn*10+a;
        n=n/10;
        }
        if(b==rn)
        {
        cout<<"palindrome";
        }
        else
        {
        cout<<"not palindrome";
        }
}
```

Example 1.20 Write a program to check whether a no is Armstrong number or not. (A number is said to be Armstrong number if sum of cube of its digit is equal to that number).

Solution:

```
#include<iostream.h>
void main()
{
        int a, n,s=0, b;
        cout<<"Enter a number";
        cin>>n;
        b=n;
        while(n!=0)
        {
        a=n%10;
        s=s+a*a*a;
        n=n/10;
        }
        if(b==s)
        {
        cout<<"Armstrong";
        }
        else
        {
        cout<<"Not Armstrong";
        }
}
```

Example 1.21 Write a program to print all the prime numbers from 1 to 500.

Solution:

```
#include<iostream.h>
void main( )
{
        int n, i ;
        for(n=2; n<=500; n++)
        {
                for(i=2; i<=n-1; i++)
                {
                        if ( n% i == 0 )
                        {
                        break ;
```

```
                              }
                    }
                    if ( i == n )
                    {
                              cout<<n<< "\t" ;
                    }
          }
}
```

Array

Collection of similar data types stored in contiguous memory location is known as array.
Syntax is:**Data_type array_name[size];**

i.e. int a[20]; means a is an array which can hold 20 integers. char nm[16]; means nm is an array which can hold 16 character. An array index always starts from 0 and index of last element is n-1, where n is the size of the array.

In the above example:

a[0] is the first element.

a[1] is the second element.............

And a[19] is the last element of the array a.

Structure

A structure is a collection of dissimilar data types.

```
struct book
{
        char name ;
        float price ;
        int pages ;
};
struct book b1, b2, b3 ;
```

Here b1, b2 and b3 are structure variable of type book. And name, price and pages are called structure members. To access a structure member we need dot(.) operator.

b1.name - name of book b1.

b1.price – price of book b1.

b1.pages – price of book b1.

Union

A union is a collection of dissimilar datatypes.

```
union book
{
        char name ;
        float price ;
        int pages ;
} ;
unionbook b1, b2, b3 ;
```

Here b1, b2 and b3 are union variable of type book.

Here b1, b2 and b3 are union variable of type book. And name, price and pages are called union members. To access a union member we need dot(.) operator.

b1.name - name of book b1.

b1.price – price of book b1.
b1.pages – price of book b1.

Difference between structure and union

Structure	Union

1) Syntax:
struct structure_name
{
//Data types
}
	1) Syntax:
	union union_name
	{
	//Data types
	}

2) All the members of the structure can be accessed at once.	2) In a union only one member can be used at a time.

3) Structure allocates the memory equal to the total memory required by the members.
struct example{
 int x;
 float y;
}
Here memory allocated is size of(x)+sizeof(y).
	3) Union allocates the memory equal to the maximum memory required by the member of the union.
	union example{
	 int x;
	 float y;
	}
	Here memory allocated is sizeof(y), because size of float is more than size of integer.

Function

A function is a self-contained block of statements that perform a coherent task of some kind. Basically a function consists of three parts:

Function prototype: Function prototype is the skeleton of a function.

Syntax is: return_type function_name(type of arguments);

Example: int add(int, int);   void display(void);

Function Calling: A function gets called when the function name is followed by a semicolon.

Syntax is : function_name(list of arguments);

Example: sum(x,y);   search(mark);

Function Definition: A function is defined when function name is followed by a pair of braces in which one or more statements may be present.

Syntax is:
Return_type function_name(list of arguments with their types)
{
//Function Body
}

Call by value and call by reference

- In call by value, value of the argument is passed during function calling. In call by reference address of the argument is passed during function calling.
- In call by value of actual arguments do not changes. But in call by reference value of actual argument changes.
- Memory can be saved if we call a function by reference.

Program 1.22 **Write a program to swap two integers using call by value and call by reference.**

Solution:

Call by value	Call by refernece

``#include<iostream.h>
void swap(int, int);
void main()
{
        int x=5, y=7;
        swap(x, y);
        cout<<x<<y<<endl;
}
void swap(int a, int b)
{
        int t=a;
        a=b;
        b=t;
        cout<<a<<b;```
	``#include<iostream.h>
	void swap(int *, int *);
	void main()
	{
	        int x=5, y=7;
	        swap(&x, &y);
	        cout<<x<<y<<endl;
	}
	void swap(int *a, int *b)
	{
	        int t=*a;
	        *a=*b;
	        *b=t;
	        cout<<*a<<*b;```

|	
}	}
Output:	**Output:**
5 7	7 5
7 5	7 5

Default arguments

- C++ allows us to call a function without specifying all its arguments.
- In such cases function assigns a default value to the parameter which does not have a matching argument in the function call.
- Default values are specified when the function is declared.

Consider the following function prototype:

float amount(**int** principal, **int** period, **float** rate=0.15);

We can call the above function by amount (5000, 7);. Here default value 0.15 will be assigned to the third argument. **We must add default values from right to left.**

```
int Add(int i, int j=5, int k=10);        // legal
int Add(int i=5, int j);                  // illegal
int Add(int i=0, int j, int k=10);        // illegal
int Add(int i=2, int j=5, int k=10);      // legal
```
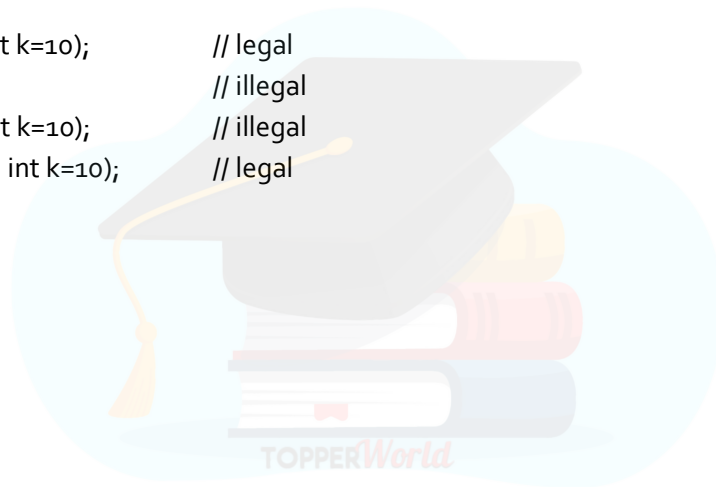
Short Type Questions

1. What is a structure and how to define and declare a structure?
2. What are the advantages of using unions?
3. Can we initialize unions?
4. Why can't we compare structures?
5. Define array and pointer?
6. Define keyword.
7. Differentiate between = and ==.
8. Differentiate between while loop and do while loop.
9. Define loop. Write the syntax of for loop.
10. Differentiate between **const** and **volatile.**

Long Type Questions

1. Why structure is used instead of union? Write the difference between structure and union.
2. What is the difference between procedure oriented programming and object oriented programming?
3. Describe the basic concepts of object oriented programming briefly.
4. Describe different types of operators in c++.
5. What is pointer? Differentiate between call by value and call by reference.
6. Explain function prototype, function calling and function definition with suitable examples.
7. Write a program to find factorial of a number.
8. Write a program to find largest among a list of integers stored in an array.
9. Using structure, write a program to find addition of two complex number.
10. Write a program to check whether a number is Armstrong or not?
11. Write a program to reverse a number using function recursion.
12. Write both recursive and non recursive function to find gcd of two integers.
13. Write a program to print fibonacii sequence up to 50 terms. (Assume, first two terms of fibonacii sequence are 0 and 1 respectively.)
14. Write a program to print all integers from 1 to n using recursion, where the value of n is supplied by the user.
15. Write a program to convert a 2×2 matrix into 3×3 matrix, where a new row element is obtained by adding all elements in that row, a new column element is obtained by adding all elements in that column and the diagonal element is obtained by summing all diagonal elements of given 2×2 matrix.

Class and Object

A class is a way to bind the data and its associated functions together. It allows the data and functions to be hidden, if necessary, from external use. A class declaration is similar syntactically to a structure.

General form of a class declaration is:

class class_name
{
private:
Variable declaration/data members;
Function declaration/ member functions;
 	
protected:
Variable declaration/data members;
Function declaration/ member functions;
 	
public:
Variable declaration/data members;
Function declaration/ member functions;
};
	Private members can be accessed only from within the class.
	
	Protected members can be accessed by own class and its derived classes.
	
	Public members can be accessed from outside the class also.

Points to remember!
- ✓ The variables declared inside the class definition are known as **data members** and the functions declared inside a class are known as **member functions**.
- ✓ Wrapping of data and function and function into a single unit (i.e. class) is known as **data encapsulation**.
- ✓ By default the data members and member function of a class are **private**.
- ✓ Private data members can be accessed by the functions that are wrapped inside the class.

General steps to write a C++ program using class and object:

- ▪ Header files
- ▪ Class definition
- ▪ Member function definition
- ▪ void main function

Program 2.1 Write a program to find sum of two integers using class and object.

Solution:

#include<iostream.h>

```cpp
class Add

{
        int x, y, z;
        public:
        void getdata()
        {
        cout<<"Enter two numbers";
        cin>>x>>y;
        }
        void calculate(void);
        void display(void);
};

void Add :: calculate()
{
        z=x+y;
}

void Add :: display()
{
        cout<<z;
}

void main()
{
        Add a;
        a.getdata();
        a.calculate();
        a.display();
}
```

Output:
Enter two numbers 5   6
11

A member function can be defined:

    (i)   **Inside** the class definition
    (ii)   **Outside** side the class definition using scope resolution operator (::).

- Here in the above example we are defining the member function getdata() inside the class definition. And we are defining the member functions calculate() and display(), outside the class definition using the scope resolution operator.
- Here void Add :: calculate() means the scpoe of member function calculate() is inside the class Add or we can say the function calculate() belongs to the class Add. **::** is the scope resolution operator which tells the scope of a member function.
- We cannot directly call a function, we can call it using object (through . operator) of the class in which the function is declared.

How to access member of a class?

To access member of a class dot operator is used. i.e.

object-name**.**data-member and

object-name**.**member-function

Application of Scope resolution operator (::)
- It is used to specify scope of a member function.
- It is used to access a global variable.

Object as Function argument

Program 2.2 **Write a program to add two time objects (in the form hh:mm).**

Solution:

```
#include<iostream.h>
class time
{
        int hours, minutes;

        public:
        void gettime(int h, int m)
        {
        hours=h;
        minutes=m;
        }
        void sum(time, time);
        void display(void);
        };
        void  time :: sum (time t1, time t2)
        {
        minutes=t1.minutes+t2.minutes;
        hours=minutes/60;
```

```cpp
            minutes=minutes%60;
            hours=hours+t1.hours+t2.hours;
            }
            void time :: display()
            {
            cout<<hours<<" : "<<minutes<<endl;
}

void main()
{
            time T1, T2, T3;
            T1.gettime(2,45);
            T2.gettime(3,30);
            T3.sum(T1, T2);
            T1.display();
            T2.display();
            cout<<"Addition of above two time is ";
            T3.display();
}
```

Output:

2 : 45

3 : 15

Addition of above two time is 6:15

Array of object

Collection of similar types of object is known as array of objects.

Program 2.3   **Write a program to input name and age of 5 employees and display them.**

Solution:
```cpp
#include<iostream.h>
class Employee
{
            char name[30];
            int age;
            public:
            void getdata(void);
            void putdata(void);
};
```

```cpp
void Employee:: getdata(void)
{
cout<<"Enter Name and Age:";
cin>>name>>age;
}

void Employee:: putdata(void)
{
cout<<name<<"\t"<<age<<endl;
}

void main()
{
        Employee e[5];
        int i;
        for(i=0; i<5; i++)
        {
                e[i].getdata();
        }
        for(i=0; i<5; i++)
        {
                e[i].putdata();
        }
}
```

Output:

Enter Name and Age: Rajib   25
Enter Name and Age: Sunil   27
Enter Name and Age: Ram   23
Enter Name and Age: Bibhuti   26
Enter Name and Age: Ramani   32
Rajib   25
Sunil   27
Ram   23
Bibhuti   26
Ramani   32

Constructor

- A constructor is a special member function whose task is to initialize the object of a class.
- Its name is same as the class name.
- A constructor does not have a return type.

- A constructor is called or invoked when the object of its associated class is created.
- It is called constructor because it constructs the values of data members of the class.
- A constructor cannot be virtual (shall be discussed later on).
- A constructor can be overloaded.

There three types of constructor:

(i)   Default Constructor
(ii)   Parameterized Constructor
(iii)   Copy constructor

Default Constructor

The constructor which has no arguments is known as default constructor.

Program 2.4   **Demonstration of default Constructor.**

Solution:
```cpp
#include<iostream.h>
class Add
{
        int x, y, z;

        public:
        Add();                          // Default Constructor
        void calculate(void);
        void display(void);
};
Add::Add()
{
        x=6;
        y=5;
}
void Add :: calculate()
{
        z=x+y;
}
void Add :: display()
{
        cout<<z;
}
void main()
{
        Add a;
        a.calculate();
```

```
        a.display();
}
```

Output:

11

Note: Here in the above program when the statement Add a; will execute (i.e. object is created), the default constructor Add () will be called automatically and value of x and y will be set to 6 and 5 respectively.

Parameterized constructor

The constructor which takes some argument is known as parameterized constructor.

Program 2.5 Write a program to initialize two integer variables using parameterized constructor and add them.

Solution:
```
#include<iostream.h>
class Add
{
        int x, y, z;
        public:
        Add(int, int);
        void calculate(void);
        void display(void);
};
Add :: Add(int a, int b)
{
        x=a;
        y=b;
}
void Add :: calculate()
{
        z=x+y;
}
void Add :: display()
{
        cout<<z;
}
void main()
{
```

```
        Add a(5, 6);
        a.calculate();
        a.display();
}
```
Output:

11

Note: Here in the above program when the statement Add a(5, 6); will be executed (i.e. object creation), the parameterized constructor Add (int, int) will be called automatically and value of x and y will be set to 5 and 6respectively.

A parameterized constructor can be called:

   (i) **Implicitly:** Add a(5, 6);
   (ii) **Explicitly :**Add a=Add(5, 6);

If the constructor has one argument, then we can also use object-name=value-of-argument; instead of object-name (value-of-argument); to initialize an object.

What is Dynamic Initialization of an object?

The initialization of an object at the time of execution of program is known as dynamic initialization of an object. It is achieved by parameterized constructor.

Copy Constructor

The constructor which takes reference to its own class as argument is known as copy constructor.

Program 2.6 Write a program to initialize two integer variables using parameterized constructor. Copy given integers into a new object and add them.

Solution:
```
#include<iostream.h>
class Add
{
        int x, y, z;
        public:
        Add()
        {
        }
        Add(int a, int b)
        {
        x=a;
        y=b;
```

```cpp
        }
        Add(Add &);
        void calculate(void);
        void display(void);
};

Add :: Add(Add &p)
{
        x=p.x;
        y=p.y;
        cout<<"Value of x and y for new object: "<<x<<" and "<<y<<endl;
}
void Add :: calculate()
{
        z=x+y;
}
void Add :: display()
{
        cout<<z;
}
void main()
{
        Add a(5, 6);
        Add b(a);
        b.calculate();
        b.display();
}
```

Output:
Value of x and y for new object are 5   and 6
11

Note: Here in the above program when the statement Add a(5, 6); will execute (i.e. object creation), the parameterized constructor Add (int, int) will be called automatically and value of x and y will be set to 5 and 6respectively. Now when the statement Add b(a) ; will execute, the copy constructor Add(Add&) will be called and the content of object a will be copied into object b.

What is Constructor Overloading?
If a program contains more than one constructor, then constructor is said to be overloaded.

Destructor

- It is a special member function which is executed automatically when an object is destroyed.
- Its name is same as class name but it should be preceded by the symbol ~.
- It cannot be overloaded as it takes no argument.
- It is used to delete the memory space occupied by an object.
- It has no return type.
- It should be declared in the public section of the class.

Program 2.7 **Demonstration of Destructor.**

Solution:

```cpp
#include<iostream.h>
class XYZ
{
        int  x;
        public:
        XYZ( );
        ~XYZ( );
        void display(void);
};
XYZ::XYZ( )
{
        x=9;
}

XYZ:: ~XYZ( )
{
        cout<<"Object is destroyed"<<endl;
}
void XYZ::display()
{
        cout<<x;
}
void main()
{
        XYZ xyz;
        xyz.display();
}
```

Output:

9
Object is destroyed.

Inline function

In C++, we can create short functions that are not actually called, rather their code is expanded in line at the point of each invocation. This process is similar to using a function-like macro. To cause a function to be expanded in line rather than called, precede its definition with the inline keyword.

- A function which is expanded in a line when it is called is called inline function.
- It executes faster than other member function.
- It can be recursive.
- Its body does not contain if else, switch, loop, goto statement.
- The **inline** keyword is preceded by function definition.

Why inline function is used?

Whenever a function is called, control jumps to definition part of the function. During this jumping of control, a significant amount of time is required. For functions having short definition if it is called several time, huge amount of time will be lost. Therefore we declare such function as inline so that when the function is called, rather than jumping to the definition of function, function definition is expanded in a line wherever it is called.

Program 2.8 Write a program to find area of a circle using inline function.

Solution:
```
#include<iostream.h>
inline float area(int);
void main()
{
int r;
cout<<" Enter the Value of r: ";
cin>>r;
cout<<" Area is: " << area(r);
}
inline float area (int a)
{
return(3.14*a*a);
}
```

Output:
Enter the Value of r:
7
153.86
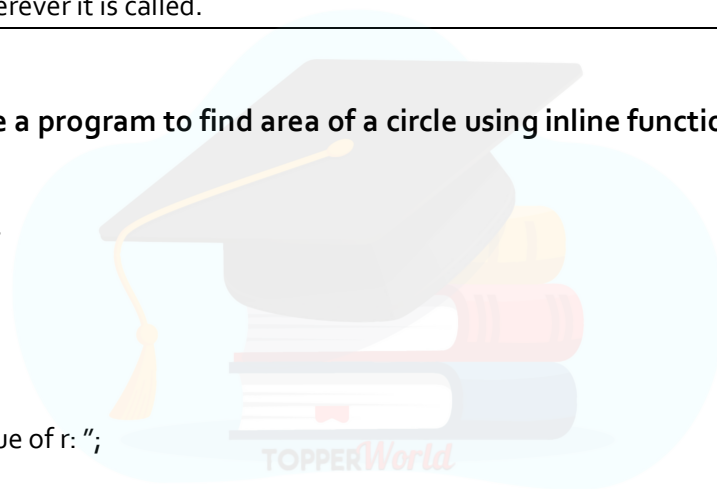
Friend Function

- Scope of a friend function is not inside the class in which it is declared.
- Since its scope is not inside the class, it cannot be called using the object of that class
- It can be called like a normal function without using any object.
- It cannot directly access the data members like other member function and it can access the data members by using object through dot operator.
- It can be declared either in private or public part of the class definition.
- Usually it has the objects as arguments.

Program 2.9 Demonstration of Friend Function.

Solution:

```cpp
#include<iostream.h>
class Add
{
        int x, y, z;
        public:
        Add(int, int);
        friend int calculate(Add p);
};

Add :: Add(int a, int b)
{
        x=a;
        y=b;
}
int calculate(Add p)
{
        return(p.x+p.y);
}
void main()
{
        Add a(5, 6);
        cout<<calculate(a);
}
```

Output:

11

Note: Here the function calculate () is called directly like normal function as it is declared as friend.

Friend Classes

It is possible for one class to be a **friend** of another class. When this is the case, the **friend** class and all of its member functions have access to the private members defined within the other class.

```cpp
#include <iostream.h>
class TwoValues
{
        int a;
        int b;
        public:
        TwoValues(int i, int j)
         {
        a = i;
        b = j;
        }
        friend class Min;
};
class Min
{
        public:
        int min(TwoValues x);
};
int Min::min(TwoValues x)
{
        return  x.a < x.b ? x.a : x.b;
}
int main()
{
        TwoValues ob(10, 20);
        Min m;
        cout << m.min(ob);
        return 0;
}
```
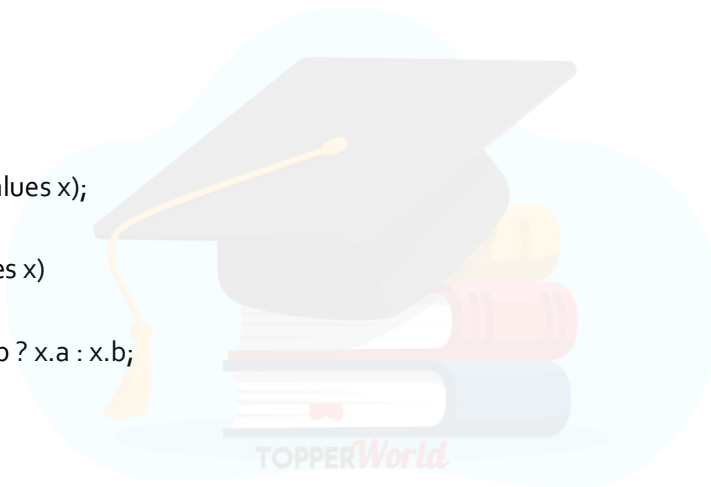
Output:
10

Note: In this example, class Min has access to the private variables a and b declared within the TwoValues class.

Static Data Members

- The data member of a class preceded by the keyword static is known as static member.
- When we precede a member variable's declaration with **static**, we are telling the compiler that only one copy of that variable will exist and that all objects of the class will share that variable. Hence static variables are called class variables.
- Unlike regular data members, individual copies of a static member variable are not made for each object. No matter how many objects of a class are created, only one copy of a **static** data member exists. Thus, all objects of that class use that same variable.
- All **static** variables are initialized to zero before the first object is created.
- Normal data members are called object variable but static data members are called class variables.

Program 2.11 Demonstration of static data members.

Solution:

```cpp
#include<iostream.h>
class A
{
        int p;
        static int q;
        public:
        A();
        void  incr(void);
        void display(void);
};
A :: A()
{
        p=5;
}
int A:: q=10;
void A:: incr()
{
        p++;
        q++;
}
void A:: display()
{
        cout<<p<<"\t"<<q<<endl;
}
void main()
```

```
{
        A a1, a2, a3;
        a1.incr();
        a1.display();
        a2.incr();
        a2.display();
        a3.incr();
        a3.display();
}
```

Output:

```
6       11
6       12
6       13
```

Note: Here p is a normal variable, whose value is 5 for all 3 objects a1, a2 and a3 (For each object, separate copy of p exists). But q is static variable or member, whose initial value is 10 and a single copy of q exists for all the objects.

Static Member function/method

- A static function can have access to only other static members (functions or variables) declared in the same class. (Of course, global functions and data may be accessed by static member functions.)
- It is accessed by class name and not by object's name i.e. class-name::function-name;
- The function name is preceded by the keyword **static**.
- A static member function does not have **this** pointer.
- There cannot be a static and a non-static version of the same function.
- A static member function may not be virtual.
- Finally, they cannot be declared as const or volatile.

Program 2.12 Demonstration of static member function.

Solution:

```
#include<iostream.h>

class ABC
{
        public:
        static int add(int, int);
};
int ABC:: add(int a, int b)
{
```

```
        return(a+b);
}

void main()
{
        ABC abc;
        int res;
        res=ABC :: add(30, 40);
        cout<<res;
}
```

Output:
70

Short Type Questions

1. What is friend function? State its properties.
2. What is inline function?
3. What is static data members and static member function?
4. How a member function of a class can be accessed?
5. Differentiate between macros and inline function.
6. What are data members and member function?
7. Define copy constructor with an example.
8. Define default constructor with an example.
9. Define dynamic initialization of object. How it is achieved?
10. What is scope resolution operator? State any two applications.

Long Type Questions

1. Explain function prototype, calling and definition with suitable examples.
2. What do you mean by static data members? Discuss.
3. WAP to find greatest of 2 no's using class and object
4. WAP to add two time objects (in the form of hh : mm :ss).
5. WAP to add two string object. The string object is initialized by following constructor. string (char[]).
6. Define a class student with member variables as roll number and name. Generate an object and initialize its variables using constructors and display them.
7. What is the difference between public, private and protected members of class?
8. What is a constructor? Explain about copy constructor with a suitable example.
9. What is a destructor? Explain with an example.
10. What is class and object? How a class differs from a structure?
11. Define a complex number. Write a program to read and print a complex number using class and object.
12. What is default argument? Discuss with a suitable example.
13. A function can return more than one value. Explain.
14. What is the difference between method overriding and method overloading? Explain your answer with suitable example.
15. How does an inline function differ from a preprocessor Macro?
16. Create a class called Employee which contains protected attributes such as emp_id, emp_salary and emp_da. emp_da is 20% of the emp_salary. Provide an appropriate method to take user input to initialize the attributes and display the details regarding 25 students of a class.
17. Write a complete program to create a class called **Account** with protected attributes such as account number and balance. The attributes should be initialized through constructors. The class contains a public method named as show () to display the initialized attributes. Provide a mechanism to create an array of **Account** objects. The array size should be given by the user at run time.
18. What is a **copy constructor**? Explain the role of a **copy constructor** while initializing a pointer attribute of a class for which the memory allocation takes place at the run time.

Find -rors (if any) and correct them:

1.
```
class My_College
{
        float cgpa;
        public:
        float My_College();
};
```

2.
```
class ABCD
{
        public:
        int INT()
        {
        return(1);
        }
};
void main()
{
        cout<<INT();
}
```

3.
```
class F
{
        int p;
        public:
        friend void print(){ }
};
void main()
{
        print();
}
```

4.
```
class M
{
        char nm[50];
        public:
        show();
};
```

Find Output:

1.
```
#include<iostream.h>
int x=5;
void main()
{
        int x=4;
        cout<<x<<::x;
}
```

2.
```
#include<iostream.h>
void main()
{
        int a=2, b=1;
        char x=1, y=0;
        if(a, b, x, y)
        cout<<"Congratulations!!";
}
```

3.
```
#include<iostream.h>
int x=5;
void main()
{
(5/2)? cout<<"Hi":cout<<"Hello";
}
```

4.
```
#include<iostream.h>
void main()
{
        int a, b;
        a=(b=7, b+2);
        cout<<a;
}
```

Inheritance

It is the process by which object of one class acquires the properties of object of another class. The class from which properties are inherited is called **base class** and the class to which properties are inherited is called **derived class**. Inheritance can be broadly classified into:

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

Base-Class Access Control

When a class inherits another, the members of the base class become members of the derived class. Class inheritance uses this general form:

```
class derived-class-name : access base-class-name
{
        // body of class
};
```

The access status of the base-class members inside the derived class is determined by access. The base-class access specifier must be either **public, private,** or **protected**. If no access specifier is present, the access specifier is **private** by default if the derived class is a **class**. If the derived class is a **struct**, then **public** is the default in the absence of an explicit access specifier.

- When the access specifier for a base class is **public**, all public members of the base become public members of the derived class, and all protected members of the base become protected members of the derived class.

- When the base class is inherited by using the **private** access specifier, all public and protected members of the base class become private members of the derived class.

- When a base class' access specifier is **protected**, public and protected members of the base become protected members of the derived class.

In all cases, the base's private elements remain private to the base and are not accessible by members of the derived class.

In a single inheritance the derived class is derived from a single base class.
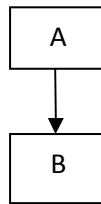
(Single inheritance)

Program 4.1   **Example of single inheritance with base class access control as public.**

Solution:
```
#include <iostream.h>
class A
{
        int i, j;
        public:
        void set(int a, int b)
        {
        i=a; j=b;
        }
        void show()
        {
         cout << i << " " << j << "\n";
        }
};
class B : public A
{
        int k;
        public:
        B(int x)
        {
         k=x;
        }
        void showk()
        {
        cout << k << "\n";
        }
};
```

```cpp
void main()
{
        B b(3);
        b.set(1, 2);
        b.show();
        b.showk();
}
```

Output:
```
1 2
3
```

Note: Here all public and protected members of the base class become private members of the derived class. So object of derived class cannot directly access the member function and data members of the base class.

Program 4.2 Example of single inheritance with base class access control as private.

Solution:

```cpp
#include <iostream.h>
class A
{
        int i, j;
        public:
        void set(int a, int b)
        {
        i=a; j=b;
        }
        void show()
        {
         cout << i << " " << j << "\n";
        }
};

class B : private A
{
        int k;
        public:

        B(int x)
        {
         k=x;
        }
        void showk()
        {
        cout << k << "\n";
        }
```

```
};

void main()
{
        B b(3);
        b.set(1, 2);
        b.show(); //******-ror******
}
```

How to access the private data member of base class in derived class?
Private data members of base class can be accessed by derived class by using public member function/methods of the base class.

Multiple Inheritance

In multiple inheritance derived class is derived from more than one base class.
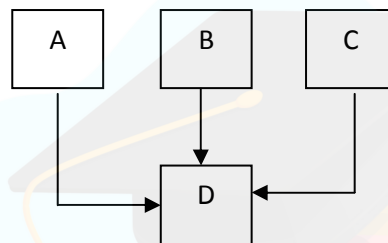
(Multiple Inheritance)

Multilevel Inheritance

In multilevel inheritance class B is derived from a class A and a class C is derived from the class B.

Syntax:
class base-class-name1
{
Data members
Member functions
};

class derived-class-name : **visibility mode** base-class-name
{

Data members
Member functions

```
};

class derived-class-name1: visibility mode derived-class-name
{
Data members
Member functions
};
```

Note: visibility mode can be either private, public or protected
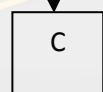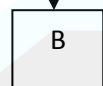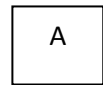
(Multilevel inheritance)

Hierarchical Inheritance

In hierarchical inheritance several classes can be derived from a single base class

```
Syntax:
class base-class-name
{
Data members
Member functions
};

class derived-class-name1 : visibility mode base-class-name
{

Data members
Member functions

};
```

```
class derived-class-name2: visibility mode base-class-name
{
Data members
Member functions
};
```
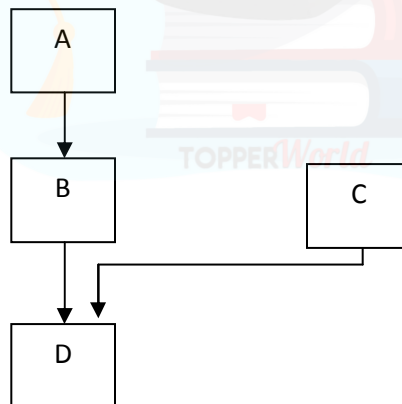
Note: visibility mode can be either private, public or protected
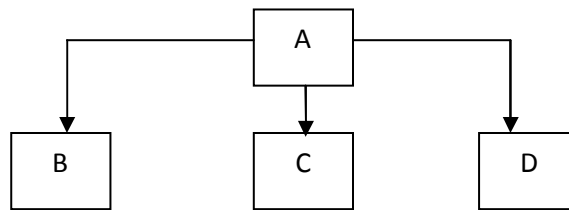
(Hierarchical inheritance)

Hybrid inheritance

It is the mixture of one or more above inheritance.

Constructor and Destructor Execution in Inheritance

When an object of a derived class is created, if the base class contains a constructor, it will be called first, followed by the derived class' constructor. When a derived object is destroyed, its destructor is called first, followed by the base class' destructor, if it exists (i.e. constructor functions are executed in their order of derivation. Destructor functions are executed in reverse order of derivation).

Solution:

```cpp
#include <iostream.h>
class base
{
        public:
        base()
        {
        cout << "Constructing base\n";
        }
        ~base()
        {
        cout << "Destructing base\n";
        }
};
class derived: public base
{
        public:
        derived()
        {
        cout << "Constructing derived\n";
        }
        ~derived()
        {
        cout << "Destructing derived\n";
        }
};
void main()
{
        derived ob;
}
```

Output:

Constructing base
Constructing derived
Destructing derived
Destructing base

Note: In the above program, first base's constructor is executed followed by derived's. Next (because ob is immediately destroyed in this program), derived's destructor is called, followed by base's.

Solution:

```cpp
#include <iostream.h>
class base
{
        public:
        base()
        {
        cout << "Constructing base\n";
        }
        ~base()
        {
        cout << "Destructing base\n";
        }
};
class derived1 : public base
{
        public:
        derived1()
        {
        cout << "Constructing derived1\n";
        }
        ~derived1()
        {
        cout << "Destructing derived1\n";
        }
};
class derived2: public derived1
{
        public:
        derived2()
        {
        cout << "Constructing derived2\n";
        }
        ~derived2()
        {
        cout << "Destructing derived2\n";
        }
};
void main()
{
```
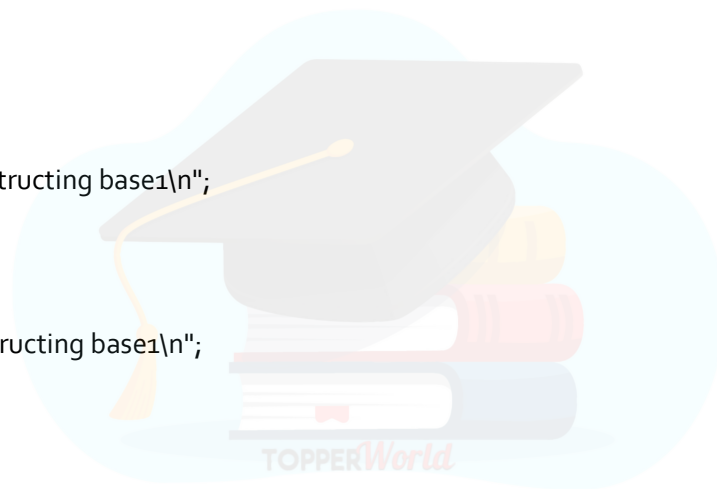
```
        derived2 ob;
}
```

Output:
Constructing base
Constructing derived1
Constructing derived2
Destructing derived2
Destructing derived1
Destructing base

Program 4.6 Constructor and Destructor execution in multiple inheritance.

Solution:

```
#include <iostream.h>
class base1
{
        public:
        base1()
        {
        cout << "Constructing base1\n";
        }
        ~base1()
        {
         cout << "Destructing base1\n";
        }
};
class base2
{
        public:
        base2()
        {
        cout << "Constructing base2\n";
        }
        ~base2()
        {
        cout << "Destructing base2\n";
        }
};
class derived: public base1, public base2
{
        public:
```

```
        derived()
        {
        cout << "Constructing derived\n";
        }
        ~derived()
        {
        cout << "Destructing derived\n";
        }
};
void  main()
{
        derived ob;
}
```

Output:
Constructing base1
Constructing base2
Constructing derived
Destructing derived
Destructing base2
Destructing base1

Note: In the above program, constructors are called in order of derivation, left to right, as specified in derived's inheritance list. Destructors are called in reverse order, right to left.

Passing Parameters to Base-Class Constructors

So far, none of the preceding examples have included constructor functions that require arguments. In cases where only the derived class' constructor requires one or more parameters, we simply use the standard parameterized constructor syntax. However, how do you pass arguments to a constructor in a base class? The answer is to use an expanded form of the derived class's constructor declaration that passes along arguments to one or more base-class constructors. The general form of this expanded derived-class constructor declaration is shown here:

```
derived-constructor (arg-list) : base1(arg-list),base2(arg-list), ......,baseN(arg-list)
{
        // body of derived constructor
}
```

Here, base1 through baseN are the names of the base classes inherited by the derived class. Notice that a colon separates the derived class' constructor declaration from the base-class specifications, and that the base-class specifications are separated from each other by commas, in the case of multiple base classes.

Program 4.7 Passing Parameters to Base Class Constructors in Single Inheritance.

Solution:

```cpp
#include <iostream.h>
class base
{
        protected:
        int i;
        public:
        base(int x)
        {
         i=x; cout << "Constructing base\n";
        }
        ~base()
        {
        cout << "Destructing base\n";
        }
};
class derived: public base
{
        int j;
        public:
        derived(int x, int y): base(y)
        {
        j=x; cout << "Constructing derived\n";
        }
        ~derived()
        {
        cout << "Destructing derived\n";
        }
        void show()
        {
        cout << i << " " << j << "\n";
        }
};
void main()
{
        derived ob(3, 4);
        ob.show();
}
```
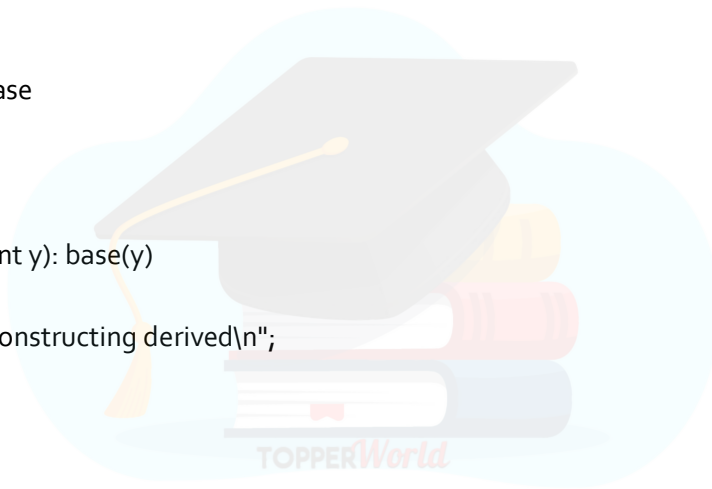
An element of ambiguity can be introduced into a C++ program when multiple base classes are inherited. For example, consider this incorrect program:
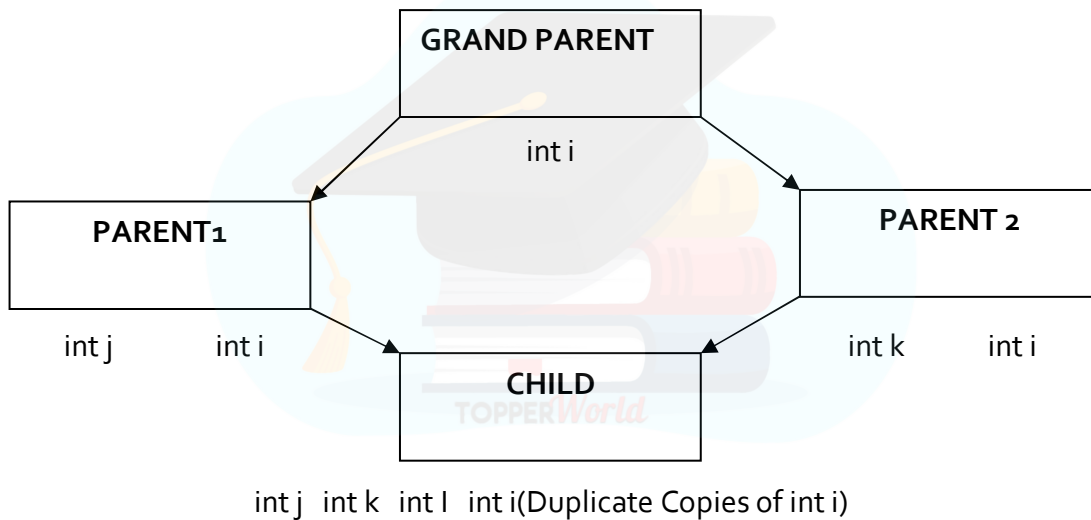
What is Multipath Inheritance?
Multipath Inheritance is a hybrid inheritance (also called as Virtual Inheritance). It is combination of hierarchical inheritance and multiple inheritance.

In Multipath Inheritance there is a one base class GRANDPARENT. Two derived class PARENT1 and PARENT2 which are inherited from GRANDPARENT. Third Derived class CHILD which is inherited from both PARENT1 and PARENT2.

Problem in Multipath Inheritance
There is an ambiguity problem. When we run program with such type inheritance, it gives a compile time error [Ambiguity]. If we see the structure of Multipath Inheritance then we find that there is shape like Diamond.

int j   int k   int I   int i(Duplicate Copies of int i)

[Ambiguity in Multipath Inheritance]

Why Ambiguity Problem in multipath (Virtual) Inheritance?
Suppose GRANDPARENT has a data member int i. PARENT1 has a data member int j. Another PARENT2 has a data member int k. CHILD class which is inherited from PARENT1 and PARENT2. CHILD class have data member:
int j (one copy of data member PARENT1)
int k (one copy of data member PARENT2)
int i(two copy of data member GRANDPARENT)

This is ambiguity problem. In CHILD class have two copies of Base class. There are two duplicate copies of int i of base class. One copy through PARENT1 and another copy from PARENT2.
This problem is also called as DIAMOND Problem.

Program 4.7 Demonstration of ambiguities in multipath inheritance.

Solution:

```
// This program contains an error and will not compile.
#include <iostream.h>
class base
{
        public:
        int i;
};
class derived1 : public base
{
        public:
        int j;
};
class derived2 : public base
{
        public:
        int k;
};
class derived3 : public derived1, public derived2
{
        public:
        int sum;
};
void  main()
{
        derived3 ob;
        ob.i = 10; // this is ambiguous, which i???
        ob.j = 20;
        ob.k = 30;
        ob.sum = ob.i + ob.j + ob.k; // i ambiguous here, too
        cout << ob.i << " "; // also ambiguous, which i?
        cout << ob.j << " " << ob.k << " ";
        cout << ob.sum;
}
```

As the comments in the program indicate, both derived1 and derived2 inherit base. However, derived3 inherits both derived1 and derived2. This means that there are two copies of base present in an object of type derived3. Therefore, in an expression like

ob.i = 10; which i is being referred to, the one in derived1 or the one in derived2? Because there are two copies of base present in object ob, there are two ob.is! As we can see, the statement is inherently ambiguous.

There are two ways to remedy the preceding program. The first is to apply the **scope resolution operator** to i and manually select one i. The second is to use **virtual base class**.

Remove Ambiguities using scope resolution operator:

In Program 4.7, the ambiguous statements

ob.i = 10;
ob.sum = ob.i + ob.j + ob.k;
cout << ob.i << " "; will be replaced by

ob.derived1::i = 10;
ob.sum = ob. derived1::i + ob.j + ob.k;
cout << ob. derived1::i << " "; respectively

As we can see, because the :: was applied, the program has manually selected derived1's version of base. However, this solution raises a deeper issue: What if only one copy of base is actually required? Is there some way to prevent two copies from being included in derived3? The answer, as you probably have guessed, is yes. This solution is achieved using virtual base classes.

Remove Ambiguities using virtual base class:

When two or more objects are derived from a common base class, we can prevent multiple copies of the base class from being present in an object derived from those objects by declaring the base class as virtual when it is inherited. We accomplish this by preceding the base class' name with the keyword virtual when it is inherited. For example, here is another version of the example program in which derived3 contains only one copy of base:

Program 4.8 Remove Ambiguities using virtual base class.

Solution:
```
#include <iostream.h>
class base
{
        public:
        int i;
};
class derived1 : virtual public base
{
```

```cpp
        public:
        int j;
};
class derived2 : virtual public base
{
        public:
        int k;
};
class derived3 : public derived1, public derived2
{
        public:
        int sum;
};
void  main()
{
        derived3 ob;
        ob.i = 10; // now unambiguous

        ob.j = 20;
        ob.k = 30;
        ob.sum = ob.i + ob.j + ob.k; // unambiguous
        cout << ob.i << " "; // unambiguous
        cout << ob.j << " " << ob.k << " ";
        cout << ob.sum;
}
```

As we can see, the keyword virtual precedes the rest of the inherited class specification. Now that both derived1 and derived2 have inherited base as virtual, any multiple inheritance involving them will cause only one copy of base to be present. Therefore, in derived3, there is only one copy of base and ob.i = 10 is perfectly valid and unambiguous.

Short Type Questions

1. Define Inheritance.
2. Write any two advantages of inheritance.
3. List various types of inheritances.
4. Define virtual base class.
5. How do the properties of the following two derived classes differ?
   class X: private Y { //....};
   class A: proctected B { //....};
6. Private attributes can't be inherited. State a remedy for this problem so that attributes of a class behave like private attributes but can be inherited. Explain with an example.

Long Type Questions

1. Describe various types of Inheritance with suitable example.
2. How to restore the access label of an inheritance data member in derived class? Explain with the help of a program in C++.
3. Give the definition of a virtual base class in C++ syntax.Explain why virtual base classes are required?
4. Explain why object-oriented programs are more maintainable and reusable compared to function-oriented programs.
5. Define ambiguity in inheritance. How ambiguities can be removed by using scope resolution operator and virtual base class. Explain your answer with example.
6. With an appropriate example, explain how ambiguities can be resolved for public and protected attributes in case of multi path inheritance without using virtual base class.

Polymorphism and Virtual Functions

Polymorphism means one name, multiple forms.

For example, the + (plus) operator in C++ will behave different for different data types:

4 + 5 <-- integer addition

3.14 + 2.0 <-- floating point addition

"Good" + "Boy" <-- string concatenation

Polymorphism is of two types:

(i)   **Compile time(Static) polymorphism** (**static binding** or **static linking** or **early binding**)
- In compile time polymorphism, all information needed to call a function is known during program compilation.
- **Example: Function overloading** and **operator overloading** are used to achieve compile time polymorphism

(ii)   **Runtime(Dynamic) polymorphism**(**late binding** or **dynamic binding**)
- In runtime polymorphism all information needed to call a function is known during program execution.
- **Example: Virtual function** is used to achieve runtime polymorphism.

Function Overloading

It is the process by which a single function can perform different task, depending upon no of parameters and types of parameters.

Program 4.1   Write a program to overload function area () to calculate area of circle and area of a rectangle.

Solution:
```
#include <iostream.h>
float area(int);
int area(int, int);
void main( )
{
        int r, l, b;
        cout << "Enter the Value of r, l & b: ";
        cin>>r>>l>>b;
        cout<< "Area of circle is "<<area(r)<<endl;
        cout<< "Area of rectangle is "<<area(l,b);
}

float area(int a)
```

```
{
        return (3.14*a*a);
}
int area(int a, int b)
{
        return (a*b);
}
```

Output:
Enter the Value of r, l & b:
7 8 6
Area of circle is 153.86
Area of circle is 48

Ambiguity in Function Overloading

Suppose we have two functions:
```
void area(int,int);
void area(float,int);
void main()
{
        area(10,10);              // Unambiguous function call, calls area(int, int){ }
        area(10.0,10);            // Ambiguous function call, error!
}
```
Note: Here, the second area() function will not compile and will generate error ambiguity between area(int,int) and area(float, int) . It's because 10.0 is treated as a double, not a float. Either of our functions could accept a double, but our compiler doesn't know which one you want to use.

Following functions cannot be overloaded:
- void f(int x);
  void f(int &x);
  Above two functions cannot be overloaded when the only difference is that one takes a reference parameter and the other takes a normal, call-by-value parameter.

- typedef int integer;
  enum days{mon,tue,wed}
  void f(int);
  void f(mon);

Object Slicing

In object-oriented programming, a derived class typically inherits its base class by defining additional member variables. If a base class instance is assigned its value from a derived class instance, member variables defined in the derived class cannot be copied, since the base class has no place to store them. This is a natural and unavoidable consequence of assignment by value from derived class objects. The

term object slicing is sometimes used to refer to this aspect of assignment by value to a base class instance.

Object Slicing occurs when object of a derived class is assigned to an object of a base class, thereby losing part of the information - some of it is "sliced" away.

Example:
```
class A
{
        int x;
};
class B : public A
{
        int y;
};
```

So an object of type B has two data members, x and y Then if we are writing this code:
```
B b;
A a = b;
```
Then the information in b about member y will be lost in a.

Function Overriding using Virtual Function

When a base class and derived class contain same member function, then the base version of the member function always works when we invoke that function using base pointer.

Program 4.2 Demonstration of function overriding.

Solution:
```
#include<iostream.h>
class B
{
        public:
        void show()
        {
        cout<<"I am in base show"<<endl;
        }
};
class D:public B
{
        public:
        void show()
        {
        cout<<"I am in derived show"<<endl;
```

```
        }
};
void main()
{
        B  b, *bp;
        D  d;
        bp=&b;
        bp->show();
        bp=&d;
        bp->show();
}
```

Output:
I am in base show
I am in base show

Note: Here the base version of function show () will work as it overrides the derived version of show ().

A **virtual function** is a member function that is declared within a base class and redefined by a derived class. To create a virtual function, precede the function's declaration in the base class with the keyword virtual. When a class containing a virtual function is inherited, the derived class redefines the virtual function to fit its own needs. In essence, virtual functions implement the "one interface, multiple methods" philosophy that underlies polymorphism. The virtual function within the base class defines the form of the interface to that function. Each redefinition of the virtual function by a derived class implements its operation as it relates specifically to the derived class. That is, the redefinition creates a specific method.

Program 4.3 **Write a program to demonstrate virtual function.**

Solution:
```
#include<iostream.h>
class B
{
        public:
        virtual void show()
        {
        cout<<"I am in base show"<<endl;
        }
};
class D:public B
{
        public:
```

```
        void show()
        {
        cout<<"I am in derived show"<<endl;
        }
};
void main()
{
        B  b, *bp;
        D  d;
        bp=&b;
        bp->show();
        bp=&d;
        bp->show();
}
```

Output:
I am in base show
I am in derived show

Important Tips!

A base pointer can be made to point to any number of derived objects, it cannot access the members defined by a derived class. It can access only the members which are common to the base class. If a same function is present in both base and derived class, always base version of the function is called when we access the function using base pointer (no matters whether it points to base class or derived class). Derived version of the function can be called by making the function (having same name) as virtual. This is also called **function overriding** because function in the base class is overridden by the function in the derived class.

Remember!

When a virtual function is inherited, its virtual nature is also inherited. This means that when a derived class that has inherited a virtual function is itself used as a base class for another derived class, the virtual function can still be overridden.

Pure virtual function

A pure virtual function is a virtual function that has no definition within the base class. To declare a pure virtual function, use this general form:

virtual return-type function-name (parameter-list) = 0;

When a virtual function is made pure, any derived class must provide its own definition. If the derived class fails to override the pure virtual function, a compile-time error will result. Hence definition for pure virtual function must be there in the derived class.

Abstract Classes

A class that contains at least one pure virtual function is said to be abstract. Because an abstract class contains one or more functions for which there is no definition (that is, a pure virtual function), no objects of an abstract class may be created.

Instead, an abstract class constitutes an incomplete type that is used as a foundation for derived classes. Although we cannot create objects of an abstract class, we can create pointers and references to an abstract class. This allows abstract classes to support run-time polymorphism, which relies upon base-class pointers and references to select the proper virtual function.

Program 4.4 **Create an abstract class called Shape which contains a pure virtual function called find_vol() and a protected attribute named as volume. Create two new derived classes from the above class named as Cube and Sphere having double type attribute named as side and radius respectively. Implement dynamic polymorphism to find out volume of a cube and a sphere. Also display the result.** **[BPUT 2010]**

Solution:
```cpp
#include<iostream.h>
class Shape
{
        protected:
        double volume;
        public:
        virtual void find_vol()=0;
};
class Cube: public  Shape
{
        protected:
        double side;
        public:
        Cube();
        void find_vol();
};

class Sphere: public  Shape
{
        protected:
        double radius;
        public:
```

```cpp
            Sphere();
            void find_vol();
};

Cube::Cube()
{
cout<<"Enter side of the Cube:"<<endl;
cin>>side;
}
Sphere::Sphere ()
{
cout<<"Enter radius of the sphere:"<<endl;
cin>>radius;
}

void Cube:: find_vol()
{
volume=side*side*side;
cout<<"Volume of Cube is: "<<volume<<endl;
}
void Sphere:: find_vol()
{
volume=(4/3)*3.14*radius*radius*radius;
cout<<"Volume of sphere is: "<<volume;
}

void main()
{
        Shape *ptr;
        Cube cube;
        Sphere sphere;
        ptr=&cube;
        ptr->find_vol();
        ptr=&sphere;
        ptr->find_vol();
}
```
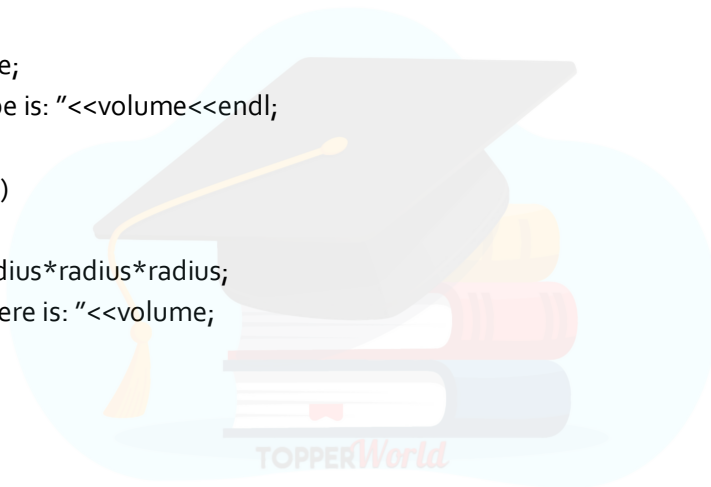
Output:
Enter side of the Cube:
3
Enter radius of the sphere:
4

Volume of Cube is: 27
Volume of sphere is: 200.96

Short Type Questions

1. Why class is called an ADT?
2. Define function overriding and function overloading.
3. Define early binding and late binding.
4. Give two example of static polymorphism.
5. What is run time polymorphism?
6. Define pure virtual function.
7. Define ambiguity in function overloading.
8. How run time polymorphism can be achieved?
9. How compile time polymorphism can be achieved?

Long Type Questions

1. Explain virtual function with a suitable example.
2. Explain function overloading with an example. Also explain ambiguity problem in function overloading.
3. Discuss object slicing with a suitable example.
4. "Pure virtual functions force the programmer to redefine virtual function inside derived class". Comment on this statement.
5. An abstract class cannot have instances. What then is the use of having abstract classes? Explain your answer using a suitable example.
6. Distinguish between virtual member function and non-virtual member function.
7. Explain function overloading and function overriding with suitable examples.
8. Create a class called Volume which contains a method called find_vol (). Write down appropriate code to create objects named as sphere and cylinder of the above class and implement function overloading to calculate volume of a sphere and cylinder based upon user input.

Operator Overloading

- It is most striking feature of C++.
- In operator overloading an operator can be operated on user defined data types. i.e. + operator perform addition of integers or real numbers. But we can overload this operator to compute sum of two complex number.
- Only existing operators can be overloaded. New operators cannot be overloaded (i.e. $ cannot be overloaded as it is not an operator.)
- It should obey the basic meaning of an operator i.e. + operator cannot be used to subtract two numbers.
- These operators cannot be overloaded:

The operator overloading can be done by using:

(i) Member function
(ii) Friend function

this Pointer

Every object in C++ has access to its own address through an important pointer called this pointer. The this pointer is an implicit parameter to all member functions. Therefore, inside a member function, this may be used to refer to the invoking object.

Friend functions do not have a **this** pointer, because friends are not members of a class. Only non static member functions have a **this** pointer.

Program 5.1 Write a program to demonstrate this pointer.

Solution:
```
#include <iostream.h>
class Box
{
        private:
        double length;
        double breadth;
        double height;

        public:
        Box(double l=2.0, double b=2.0, double h=2.0)
        {
        cout <<"Constructor called." << endl;
        length = l;
        breadth = b;
        height = h;
        }
```

```cpp
double Volume()
{
return (length * breadth * height);
}
int compare(Box box)
 {
return (this->Volume() > box.Volume());
 }

};
void main(void)
{
    Box Box1(3.3, 1.2, 1.5);
    Box Box2(8.5, 6.0, 2.0);

    if(Box1.compare(Box2))
    {
    cout << "Box2 is smaller than Box1" <<endl;
    }
    else
    {
    cout << "Box2 is equal to or larger than Box1" <<endl;
    }
  }
```
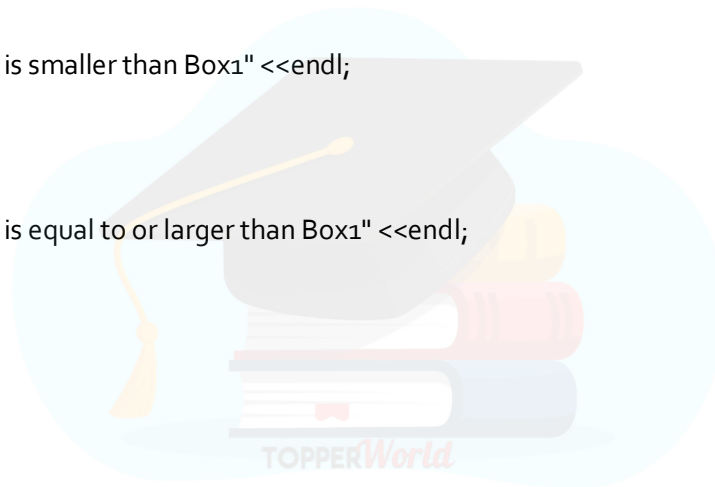
Output:
Constructor called.
Constructor called.
Box2 is equal to or larger than Box1

Overloading Operators

A member operator function takes this general form:
return_type operator symbol(arg-list)
{
// operations
}
Often, operator functions return an object of the class they operate on, but return_type can be any valid type(int, char, float, void etc). When you create an operator function, substitute the operator for the symbol. For example, if you are overloading the / operator, use operator/. When you are overloading a unary operator, arg-list will be empty. When you are overloading binary operators, arg-list will contain one parameter.

Operator Overloading Restrictions:

There are some restrictions that apply to operator overloading.

- We cannot alter the precedence of an operator.
- We cannot change the number of operands that an operator takes.
- Except for the function call, operator, operator functions cannot have default arguments.
- Finally, these operators cannot be overloaded: scope resolution operator (::), conditional operator (:?), dot operator (.) and asterisk operator (*).
- Except for the = operator, operator functions are inherited by any derived class.
- However, a derived class is free to overload any operator (including those overloaded by the base class) it chooses relative to itself.

Program 5.2 **Write a program to overload unary operator ++ using member function.**

Solution:

```
#include<iostream.h>
class A
{
        int n;
        public:
        void getdata( );
        void operator ++( );
        void display( );
};
void A::getdata( )
{
        cout<<"Enter a number";
        cin>>n;
}
void A::operator ++( )
{
        n=n+1;
}
void A::display( )
{
        cout<<n;
}
void main()
{
        A a;
        a.getdata();
        a++;
        a.display();
```

}
Output:

Enter a number

5

6

Program 5.3   Write a program to overload unary operator ++ using friend function.

Solution:
```cpp
#include<iostream.h>
class A
{
        int n;
        public:
        void getdata( );
        friend void operator ++( A &);
        void display( );
};

void A::getdata( )
{
        cout<<"Enter a number";
        cin>>n;
}
void  operator ++(A x)
{
        x.n=x.n+1;
}
void A::display( )
{
        cout<<n;
}
void main()
{
        A a;
        a.getdata();
        a++;
        a.display();
}
```
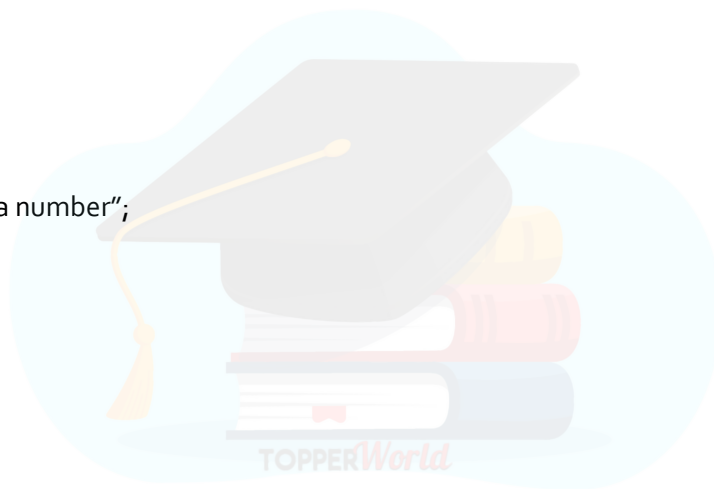
Output:

Enter a number

5
6

Program 5.4 Write a program to overload binary operator + to find sum of two complex numbers using member function.

Solution:

```cpp
#include<iostream.h>
class Complex
{
        float real, img;
        public:
        Complex(float, float );
        Complex operator + ( Complex);
        void display( );
};
Complex::Complex(float x, float y )
{
        real=x;
        img=y
}
Complex  Complex::operator +(Comlex c)
{
        Complex temp;
        temp.real=real+c.real;
        temp.img=img+c.img;
        return(temp);
}
void A::display( )
{
        cout<<real<<" +j"<<img<<"\n";
}

void main( )
{
        Complex c1(2.5, 3.4), c2(4.2, 6.5), c3;
        c3=c1+c2;
        c1.display( );
        c2.display( );
        c3.display( );
}
```
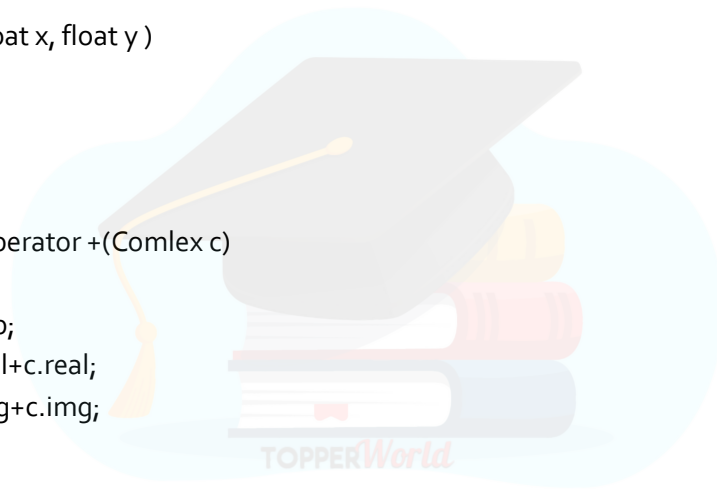
Output:
2.5+ j3.4
4.2+ j6.5
6.7+ j9.9

Note: As you can see, operator+ () has only one parameter even though it overloads the binary + operator. (You might expect two parameters corresponding to the two operands of a binary operator.) The reason that operator+ () takes only one parameter is that the operand on the left side of the + is passed implicitly to the function through the this pointer. The operand on the right is passed in the parameter c. The fact that the left operand is passed using this also implies one important point: When binary operators are overloaded, it is the object on the left that generates the call to the operator function.
The statement c3=c1+c2; is same as c1.operator+(c2).

Program 5.5 Write a program to overload binary operator + to find sum of two complex numbers using friend function.

Solution:
```
#include<iostream.h>
class Complex
{
        float real, img;
        public:
        Complex(float, float );
        friend Complex operator + ( Complex, Complex);
        void display( );
};

Complex::Complex(float x, float y )
{
        real=x;
        img=y
}

Complex  operator+(Comlex a, Complex b)
{
        Complex temp;
        temp.x=a.real+b.real;
        temp.y=a.img+b.img;
        return(temp);
}
void A::display( )
```
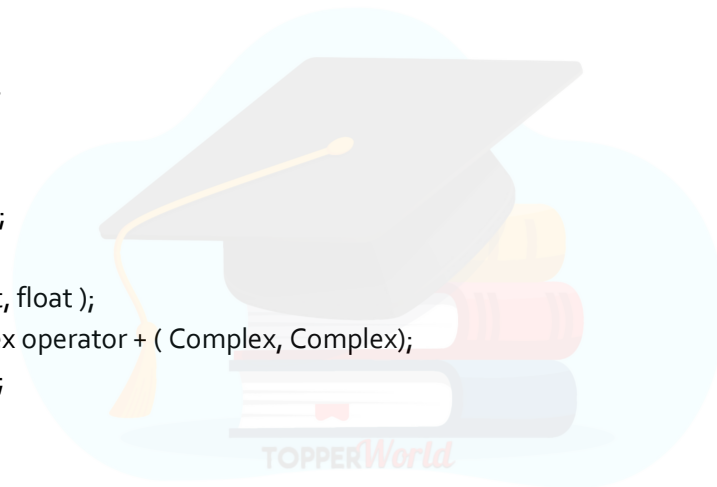
```cpp
{
        cout<<real<<" +j"<<img<<"\n";
}
void main()
{
        Complex c1(2.5, 3.4), c2(4.2, 6.5), c3;
        c3=c1+c2;
        c1.display();
        c2.display();
        c3.display();
}
```

Output:
2.5+ j3.4
4.2+ j6.5
6.7+ j9.9

Program 5.5 Suppose there is a class called X with two double type attributes. Write a C++ program to create two objects named ob 1 and ob 2 of the above class and overload the binary == operator to perform the following operation within main():
if(ob 1== ob 2)
cout<<"Objects are same"<<endl;
else
cout<<"Objects are different"<<endl;   [BPUT 2010]

Solution:
```cpp
#include<iostream.h>
class X
{
        double d1, d2;
        public:
        X(double, double);
        int operator==(X);
        void display( );
};

X::X(double x, double y )
{
        d1=x;
        d2=y;
}
int X:: operator==(X p)
```

```cpp
{
        if(d1==p.d1 &&d2==p.d2)
        return 1;
        else
        return 0;
}
void main()
{
        X ob1(2.5, 3.4), ob2(2.5, 3.0);
        if(ob 1== ob 2)
        cout<<"Objects are same"<<endl;
        else
        cout<<"Objects are different"<<endl;

}
```

Short Type Questions

1. What is operator overloading?
2. Which operators cannot be overloaded in C++ and why?

Long Type Questions

1. Create a class complex with real and imaginary parts as member variables, member function get () and disp () to input and display a complex number respectively. Write a program using the above class to overload + and – operators to perform addition and subtraction of two complex numbers.
2. Write a program in C++ to overload subscript [] operator.
3. Define a class called Increment; the class contains one integer data member. Overload the object of the class for both pre-increment and post-increment operator.
4. Write a program to compare two strings by overloading == operator.
5. Write a program to add two string using + operator overloading.
6. Write a program to overload new and delete operator.
7. Write an appropriate C++ code showing ambiguity resolving mechanism where a class attribute has same name as that of a local parameter of a member by using this pointer.
8. Write a program to overload == operator to check whether two circles are equal or not. (Two circles are said to be equal if their radius is same and center has same coordinate)
9. Write a program to overload new and delete operator in C++.
10. Write a program to overload == operator to check whether two strings are equal or not.

Exception Handling

Two common types of error in a program are:

1) **Syntax error** (arises due to missing semicolon, comma, and wrong prog. constructs etc)
2) **Logical error** (wrong understanding of the problem or wrong procedure to get the solution)
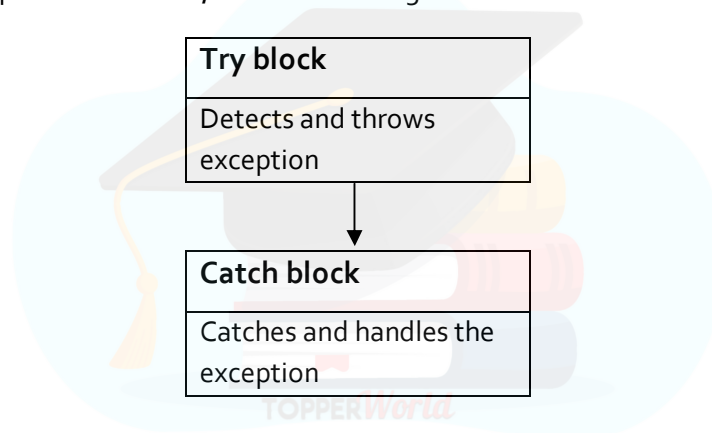
Exceptions

Exceptions are the errors occurred during a program execution. Exceptions are of two types:

- Synchronous (generated by software i.e. division by 0, array bound etc).
- Asynchronous (generated by hardware i.e. out of memory, keyboard etc).

Exception handling mechanism

- C++ exception handling mechanism is basically built upon three keywords namely, try, throw and catch.
- Try block hold a block of statements which may generate an exception.
- When an exception is detected, it is thrown using a throw statement in the try block.

Try block
Detects and throws exception

Catch block
Catches and handles the exception

- A try block can be followed by any number of catch blocks.

The general form of **try** and **catch** block is as follows:

```
try
{
        /* try block; throw exception*/
}
catch (type1 arg)
{
        /*  catch block*/
}
```

....

```
.........................
catch (type2 arg)
{
        /*  catch block*/
}
```

The exception handling mechanism is made up of the following elements:
- try blocks
- catch blocks
- throw expressions

Program 7.1 Write a program to find x/y, where x and y are given from the keyboard and both are integers.

Solution:
```
#include<iostream.h>
void main()
{
        int x, y;
        cout<<"enter two number"<<endl;
        cin>>x>>y;
        try
        {
                if(y!=0)
                {
                z=x/y;
                cout<<endl<<z;
                }
                else
                {
                throw(y);
                }
        }
        catch(int y)
        {
        cout<<"exception occurred: y="<<y<<endl;
        }
}
```

Output:
Enter two number
6 0
exception occurred:y=0

A **try** block can be localized to a function. When this is the case, each time the function is entered, the exception handling relative to that function is reset. For example, examine this program.

Program 7.2

```
#include <iostream.h>
void Xhandler(int test)
{
        try
        {
        if(test) throw test;
        }
        catch(int i)
        {
        cout << "Caught Exception #: " << i << '\n';
        }
}
void  main()
{
        cout << "Start\n";
        Xhandler(1);
        Xhandler(2);
        Xhandler(0);
        Xhandler(3);
        cout << "End";
}
```

Output:
Start
Caught Exception #: 1
Caught Exception #: 2
Caught Exception #: 3
End

As you can see, three exceptions are thrown. After each exception, the function returns. When the function is called again, the exception handling is reset.

It is important to understand that the code associated with a **catch** statement will be executed only if it catches an exception. Otherwise, execution simply bypasses the **catch** altogether. (That is, execution never flows into a **catch** statement.) For example, in the following program, no exception is thrown, so the **catch** statement does not execute.

```
#include <iostream.h>
void  main()
```

```
{
        cout << "Start\n";
        try
        {
        cout << "Inside try block\n";
        cout << "Still inside try block\n";
        }
        catch (int i)
        {
        cout << "Caught an exception -- value is: ";
        cout << i << "\n";
        }
        cout << "End";
}
```

Output:
Start
Inside try block
Still inside try block
End

Catching Class Types

An exception can be of any type, including class types that you create. Actually, in real-world programs, most exceptions will be class types rather than built-in types. Perhaps the most common reason that you will want to define a class type for an exception is to create an object that describes the error that occurred. This information can be used by the exception handler to help it process the error. The following example demonstrates this.

Program 7.3

```
#include <iostream.h>
#include <cstring.h>
class MyException
{
        public:
        char str_what[80];
        int what;
        MyException() { *str_what = 0; what = 0; }
        MyException(char *s, int e)
        {
        strcpy(str_what, s);
        what = e;
```

```
        }
};
void main()
{
        int i;
        try {
        cout << "Enter a positive number: ";
        cin >> i;
        if(i<o)
        throw MyException("Not Positive", i);
        }
        catch (MyException e) { // catch an error
        cout << e.str_what << ": ";
        cout << e.what << "\n";
        }
}
```

Output:

Enter a positive number: -4

Not Positive: -4

The program prompts the user for a positive number. If a negative number is entered, an object of the class **MyException** is created that describes the error. Thus, **MyException** encapsulates information about the error. This information is then used by the exception handler. In general, you will want to create exception classes that will encapsulate information about an error to enable the exception handler to respond effectively.

Using Multiple catch Statements

As stated, you can have more than one **catch** associated with a **try**. In fact, it is common to do so. However, each **catch** must catch a different type of exception. For example, this program catches both integers and strings.

Program 7.4

```
#include <iostream.h>
void Xhandler(int test)
{
        try
        {
        if(test) throw test;
        else throw "Value is zero";
        }
        catch(int i)
        {
```

```
        cout << "Caught Exception #: " << i << '\n';
        }
        catch(const char *str) {
        cout << "Caught a string: ";
        cout << str << '\n';
        }
}
void main()
{
        cout << "Start\n";
        Xhandler(1);
        Xhandler(2);
        Xhandler(0);
        Xhandler(3);
        cout << "End";
}
```

Output:
Start
Caught Exception #: 1
Caught Exception #: 2
Caught a string: Value is zero
Caught Exception #: 3
End

As you can see, each **catch** statement responds only to its own type.

Handling Derived-Class Exceptions

You need to be careful how you order your **catch** statements when trying to catch exception types that involve base and derived classes because a **catch** clause for a base class will also match any class derived from that base. Thus, if you want to catch exceptions of both a base class type and a derived class type, put the derived class first in the **catch** sequence. If you don't do this, the base class **catch** will also catch all derived classes. For example, consider the following program.

Program 7.5

```
#include <iostream.h>
class B
{
};
class D: public B
{
};
```

```
void main()
{
        D derived;
        try
        {
        throw derived;
        }
        catch(B b)
        {
        cout << "Caught a base class.\n";
        }
        catch(D d)
        {
        cout << "This won't execute.\n";
        }
}
```

Here, because **derived** is an object that has **B** as a base class, it will be caught by the first **catch** clause and the second clause will never execute. Some compilers will flag this condition with a warning message. Others may issue an error. Either way, to fix this condition, reverse the order of the **catch** clauses.

Exception Handling Options

There are several additional features and nuances to C++ exception handling that make it easier and more convenient to use. These attributes are discussed here.

Catching All Exceptions

In some circumstances you will want an exception handler to catch all exceptions instead of just a certain type. This is easy to accomplish. Simply use this form of **catch**.

```
catch (...) {
// process all exceptions
}
```

Here, the ellipsis matches any type of data. The following program illustrates **catch (...)**.

Program 7.6

```
#include <iostream.h>
void Xhandler(int test)
{
        try
        {
        if(test==0) throw test; // throw int
        if(test==1) throw 'a'; // throw char
        if(test==2) throw 123.23; // throw double
```

```
        }
        catch(...)
        {
        cout << "Caught One!\n";
        }
}
void main()
{
        cout << "Start\n";
        Xhandler(0);
        Xhandler(1);
        Xhandler(2);
        cout << "End";
}
```

Output:
Start
Caught One!
Caught One!
Caught One!
End

Rethrowing an Exception

If you wish to rethrow an expression from within an exception handler, you may do so by calling throw, by itself, with no exception. This causes the current exception to be passed on to an outer try/catch sequence. The most likely reason for doing so is to allow multiple handlers access to the exception. For example, perhaps one exception handler manages one aspect of an exception and a second handler copes with another. An exception can only be rethrown from within a catch block (or from any function called from within that block). When you rethrow an exception, it will not be recaught by the same catch statement. It will propagate outward to the next catch statement. The following program illustrates rethrowing an exception, in this case a char * exception.

Program 7.7
```
#include <iostream.h>
void Xhandler()
{
        try
        {
        throw "hello"; // throw a char *
        }
        catch(const char *)
        {
        cout << "Caught char * inside Xhandler\n";
        throw ; // rethrow char * out of function
```

```
        }
}
void main()
{
        cout << "Start\n";
        try
        {
        Xhandler();
        }
        catch(const char *)
        {
        cout << "Caught char * inside main\n";
        }
        cout << "End";
}
```

Output:
Start
Caught char * inside Xhandler
Caught char * inside main
end

Understanding terminate( ) and unexpected( )

As mentioned earlier, **terminate()** and **unexpected()** are called when something goes wrong during the exception handling process. These functions are supplied by the Standard C++ library. Their prototypes are shown here:

void terminate( );

void unexpected( );

These functions require the header **<exception>**.

The **terminate()** function is called whenever the exception handling subsystem fails to find a matching **catch** statement for an exception. It is also called if your program attempts to rethrow an exception when no exception was originally thrown. The **terminate()** function is also called under various other, more obscure circumstances. For example, such a circumstance could occur when, in the process of unwinding the stack because of an exception, a destructor for an object being destroyed throws an exception. In general, **terminate()** is the handler of last resort when no other handlers for an exception are available. By default, **terminate()** calls **abort()**.

Short Type Question

1. What is exception?
2. Differentiate between syntax error and logical error.

Long Type Questions

1. What is an exception? Describe the mechanism of exception handling with suitable example?
2. What is a generic catch block? What are the restrictions while using a generic catch block? Explain with an example.

Dynamic Memory Management

There are two ways to allocate memory:

    (i)   Static memory allocation

    (ii)   Dynamic memory allocation

(i) Static memory allocation

To allocate memory at the time of program compilation is known as static memory allocation.

i.e. int a[10];

it allocates 20 bytes at the time of compilation of the program. Its main disadvantage is wastage or shortage of memory space can takes place.

(ii) Dynamic memory allocation

To allocate memory at the time of program execution is known as dynamic memory allocation.C++ provides two dynamic allocation operators: **new** and **delete**. These operators are used to allocate and free memory at run time. Dynamic allocation is an important part of almost all real-world programs. These are included for the sake of compatibility with C. However, for C++ code, you should use the new and delete operators because they have several advantages. The new operator allocates memory and returns a pointer to the start of it. The delete operator frees memory previously allocated using new.

The general forms of new and delete are shown here:

p_var = new type;

delete p_var;

Here, p_var is a pointer variable that receives a pointer to memory that is large enough to hold an item of type type.

Program 7.1   **Memory Allocation to an integer.**

Solution:

```
#include <iostream.h>
void  main()
{
        int *p;
        try
        {
                p = new int;
        }
        catch (bad_alloc xa)
        {
                cout << "Allocation Failure\n";
```

```
            return 1;
        }
        *p = 100;
        cout << "At " << p << " ";
        cout << "is the value " << *p << "\n";
        delete p;
}
```

This program assigns to **p** an address in the heap that is large enough to hold an integer. It then assigns that memory the value 100 and displays the contents of the memory on the screen. Finally, it frees the dynamically allocated memory. Remember, if your compiler implements **new** such that it returns null on failure, you must change the preceding program appropriately.The **delete** operator must be used only with a valid pointer previously allocated by using **new**. Using any other type of pointer with **delete** is undefined and will almost certainly cause serious problems, such as a system crash.

Comparison between new, delete and malloc(), free

Although new and delete perform functions similar to malloc() and free(), they have several advantages. First, new automatically allocates enough memory to hold an object of the specified type. You do not need to use the sizeof operator. Because the size is computed automatically, it eliminates any possibility for error in this regard. Second, new automatically returns a pointer of the specified type. You don't need to use an explicit type cast as you do when allocating memory by using malloc(). Finally, both new and delete can be overloaded, allowing you to create customized allocation systems. Although there is no formal rule that states this, it is best not to mix new and delete with malloc() and free() in the same program. There is no guarantee that they are mutually compatible.

The Placement Forms of new and delete

There is a special form of new, called the placement form, that can be used to specify an alternative method of allocating memory. It is primarily useful when overloading the new operator for special circumstances. There is a default implementation of the placement new operator, which has this general form:

p_var = new (location) type;

Here, location specifies an address that is simply returned by new. There is also a placement form of delete, which is used to free memory allocated by the placement form of new.
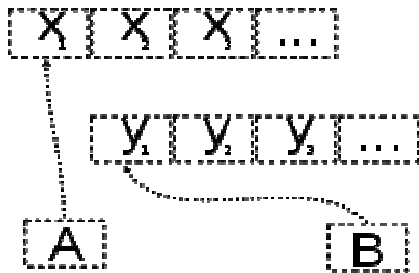
Object Copying

An **object copy** is an action in computing where a data object has its attributes copied to another object of the same data type. An object is a composite data type in object-oriented programming languages. The copying of data is one of the most common procedures that occur in computer programs. An object may be copied to reuse all or part of its data in a new context.

Methods of copying

The design goal of most objects is to give the semblance of being made out of one monolithic block even though most are not. As objects are made up of several different parts, copying becomes nontrivial. Several strategies exist to attack this problem.

Consider two objects, A and B, which each refer to two memory blocks x_i and y_i (i = 1, 2,...). Think of A and B as strings and of x_i and y_i (i = 1, 2,...) as the characters they contain. There are different strategies for copying A into B.
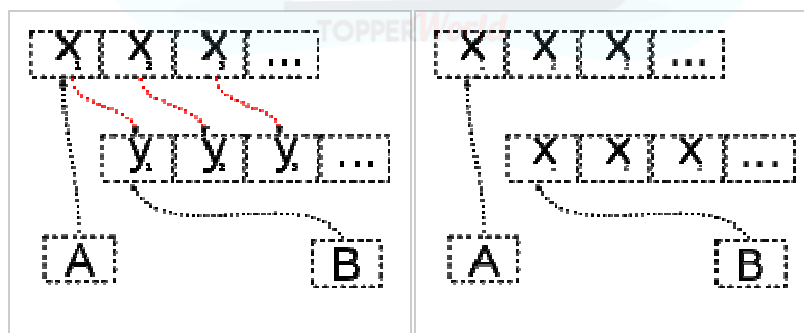

Shallow copy

One method of copying an object is the shallow copy. In the process of shallow copying A, B will copy all of A's field values. If the field value is a memory address it copies the memory address, and if the field value is a primitive type it copies the value of the primitive type.

The disadvantage is if you modify the memory address that one of B's fields point to, you are also modifying what A's fields point to.

Deep copy

A deep copy in progress.   A deep copy having been completed.

An alternative is a deep copy. Here the data is actually copied over. The result is different from the result a shallow copy gives. The advantage is that A and B do not depend on each other but at the cost of a slower and more expensive copy.

Lazy copy

A lazy copy is a combination of both strategies above. When initially copying an object, a (fast) shallow copy is used. A counter is also used to track how many objects share the data. When the program wants to modify an object, it can determine if the data is shared (by examining the counter) and can do a deep copy if necessary.

Lazy copy looks to the outside just as a deep copy but takes advantage of the speed of a shallow copy whenever possible. The downsides are rather high but constant base costs because of the counter. Also, in certain situations, circular references can cause problems. Lazy copy is related to copy-on-write.

Copy Constructor

When one object is used to initialize another, C++ performs a bitwise copy. That is, an identical copy of the initializing object is created in the target object. Although this is perfectly adequate for many cases—and generally exactly what you want to happen—there are situations in which a bitwise copy should not be used. One of the most common is when an object allocates memory when it is created. For example, assume a class called MyClass that allocates memory for each object when it is created, and an object A of that class. This means that A has already allocated its memory. Further, assume that A is used to initialize B, as shown here:

                    MyClass B= A;
If a bitwise copy is performed, then B will be an exact copy of A. This means that B will be using the same piece of allocated memory that A is using, instead of allocating its own. Clearly, this is not the desired outcome. For example, if MyClass includes a destructor that frees the memory, then the same piece of memory will be freed twice when A and B are destroyed!
The same type of problem can occur in two additional ways: first, when a copy of an object is made when it is passed as an argument to a function; second, when a temporary object is created as a return value from a function. Remember, temporary objects are automatically created to hold the return value of a function and they may also be created in certain other circumstances .To solve the type of problem just described, C++ allows you to create a copy constructor, which the compiler uses when one object initializes another. When a copy constructor exists, the default, bitwise copy is bypassed. The most common general form of a copy constructor is

classname (const classname &o)
{
// body of constructor
}

Here, o is a reference to the object on the right side of the initialization. It is permissible for a copy constructor to have additional parameters as long as they have default arguments defined for them. However, in all cases the first parameter must be a reference to the object doing the initializing.

It is important to understand that C++ defines two distinct types of situations in which the value of one object is given to another. The first is assignment. The second is initialization, which can occur any of three ways:

- When one object explicitly initializes another, such as in a declaration
- When a copy of an object is made to be passed to a function
- When a temporary object is generated (most commonly, as a return value)

The copy constructor applies only to initializations. For example, assuming a class called **myclass**, and that **y** is an object of type **myclass**, each of the following statements involves initialization.

```
myclass x = y; // y explicitly initializing x
func(y); // y passed as a parameter
y = func(); // y receiving a temporary, return object.
```

Virtual destructor

In C++ a destructor is generally used to deallocate memory and do some other cleanup for a class object and it's class members whenever an object is destroyed. Destructors are distinguished by the tilde, the '~' that appears in front of the destructor name. In order to define a virtual destructor, all you have to do is simply add the keyword "virtual" before the tilde symbol.

The need for virtual destructors in C++ is best illustrated by some examples. Let's start by going through an example that does **not** use virtual destructors, and then we will go through an example that does use virtual destructors. Once you see the difference, you will understand why virtual destructors are needed. Take a look at the code below to start out:

Program 7.2   **Example without a Virtual Destructor:**

```cpp
#include iostream.h
class Base
{
        public:
        Base(){ cout<<"Constructing Base";}
        ~Base(){ cout<<"Destroying Base";}
};

class Derive: public Base
{
    public:
        Derive(){ cout<<"Constructing Derive";}

        ~Derive(){ cout<<"Destroying Derive";}
 };

void main()
```

```
{
        Base *basePtr = new Derive();
        delete basePtr;
}
```

Output:
Constructing Base
Constructing Derive
Destroying Base

Note: Based on the output above, we can see that the constructors get called in the appropriate order when we create the Derive class object pointer in the main function. But there is a major problem with the code above: the destructor for the "Derive" class does not get called at all when we delete 'basePtr'. So, how can we fix this problem?
Well, what we can do is make the base class destructor virtual, and that will ensure that the destructor for any class that derives from Base (in our case, its the "Derive" class) will be called.

Program 7.3 **Example with a Virtual Destructor:**
So, the only thing we will need to change is the destructor in the Base class and here's what it will look like – note that we highlighted the part of the code where the virtual keyword has been added in **bold**:

```
class Base
{
        public:
        Base()
        {
        cout<<"Constructing Base";
        }
        virtual ~Base()
        {
        cout<<"Destroying Base";
        }
};
```

Output:
Constructing Base
Constructing Derive
Destroying Derive
Destroying Base
Note: Here the derived class destructor will be called before the base class. So, now you've seen why we need virtual destructors and also how they work. One important design paradigm of class design is that if a class has one or more virtual functions, then that class should also have a virtual destructor.

Short Type Questions

1. Differentiate between new and malloc.
2. Differentiate between delete and free.
3. Define object copying.
4. What is virtual destructor?
5. Can constructor be made virtual?
6. Define Copy constructor.

Long Type Questions

1. Define dynamic memory management. Explain dynamic memory management in C++ with suitable example.
2. Explain virtual destructor with an example.

Templates

- Using templates, it is possible to create generic functions and classes.
- In a generic function or class, the type of data upon which the function or class operates is specified as a parameter.
- Thus, we can use one function or class with several different types of data without having to explicitly recode specific versions for each data type.

Generic functions (Function Templates)

A generic function defines a general set of operations that will be applied to various types of data. The type of data that the function will operate upon is passed to it as a parameter. Through a generic function, a single general procedure can be applied to a wide range of data.

The general form of a template function definition is shown here:

```
template <class Ttype>
return-type func-name (Ttype a1, Ttype a2,......., Ttype n)
{
        // body of function
}
```

Here, Ttype is a placeholder name for a data type used by the function.

Program 8.1 **Write a generic function swap to interchange any two variables (integer, character, and float).**

```
#include <iostream.h>
template <class T>
void swap(T p, T q)
{
        T temp;
        temp = p;
        p = q;
        q = temp;
        cout<<p<<"\t"<<q;
}
void main()
{
        int i=10, j=20;
        float x=10.1, y=23.3;
        char a='x', b='z';
        swap (i, j);   /*swaps integers*/
        swap (x, y);   /* swaps floats*/
        swap (a, b);  /*swaps chars*/
}
```
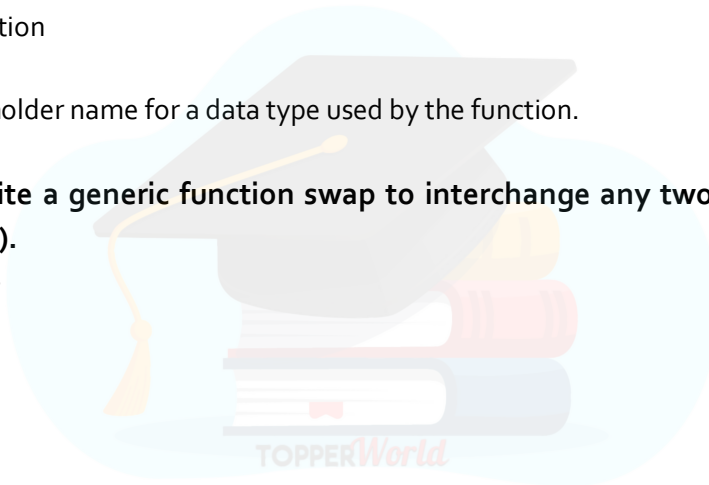
Output:

```
20      10
23.2    10.1
z       x
```

Note: The line:

template <class T> void swap (T p, T q) tells the compiler two things: that a template is being created and that a generic definition is beginning. Here, T is a generic type that is used as a placeholder. After the template portion, the function swap () is declared, using T as the data type of the values that will be swapped. In main () , the swap () function is called using three different types of data: ints, floats, and chars. Because swap () is a generic function, the compiler automatically creates three versions of swap (): one that will exchange integer values, one that will exchange floating-point values, and one that will swap characters.

A Function with Two Generic Types:

We can define more than one generic data type in the template statement by using a comma-separated list. For example; below program creates a template function that has two generic types.

Program 8.2 A function with two generic types.

Solution:

```
#include <iostream.h>
template <class T1, class T2>
void myfunc (T1 x, T2 y)
{
        cout << x << "\t" << y << "\n";
}

void main()
{
        myfunc (10, "I like C++");
        myfunc (98.6, 19);
}
```

Output:

```
10          I like C++
98.6        19
```

Generic Function Restrictions

Generic functions are similar to overloaded functions except that they are more restrictive. When functions are overloaded, you may have different actions performed within the body of each function. But a generic function must perform the same general action for all versions- only the type of data can differ.

Overloading a Function Template

In addition to creating explicit, overloaded versions of a generic function, you can also overload the template specification itself. To do so, simply create another version of the template that differs from any others in its parameter list. For example:

Program 8.2 Demonstration of function template overloading.

Solution:

```
#include <iostream.h>

// First version of f() template.
template <class X> void f(X a)
{
        cout << "Inside f(X a)\n";
}

// Second version of f() template.
template <class X, class Y> void f(X a, Y b)
{
        cout << "Inside f(X a, Y b)\n";
}
int main()
{
        f(10); // calls f(X)
        f(10, 20); // calls f(X, Y)
        return 0;
}
```

Here, the template for f() is overloaded to accept either one or two parameters

Program 8.3 Write a function template to sort an array of items.

Solution:

```
#include <iostream.h>
template <class T> void sort(X *a, int n)
{
```

```
        int i,j;
        T t;
        for(i=1; i<n;i++)
        for(j=n-1; j>=i; j--)
        if(a[j-1] > a[j])
        {
        t = a[j-1];
        a[j-1] = a[j];
        a[j] = t;
        }
}
void main()
{
        int iarray[7] = {7, 5, 4, 3, 9, 8, 6};
        double farray[5] = {2.6, -3.0, 1.2,9.6,8.9};
        int i;
        sort(iarray, 7);
        sort(farray, 5);
        cout << "Sorted INTEGER array is: ";
        for(i=0; i<7; i++)
        cout << iarray[i] << "\t";
        cout << "\nSorted CHARACTER array is: ";
        for(i=0; i<5; i++)
        cout << farray[i] << "\t";
}
```

Output:

Sorted INTEGER array is: 3   4   5   6   7   8   9

Sorted CHARACTER array is:-3.0   1.2   2.6   8.9   9.6

Generic class (class templates)

In addition to generic functions, we can also define a generic class. When we do this, we create a class that defines all the algorithms used by that class; however, the actual type of the data being manipulated will be specified as a parameter when objects of that class are created.

Generic classes are useful when a class uses logic that can be generalized. For example, the same algorithms that maintain a queue of integers will also work for a queue of characters, and the same mechanism that maintains a linked list of mailing addresses will also maintain a linked list of auto part information.

The general form of a generic class declaration is:

template <class T>

```
class class-name
{
-------------------
-------------------
};
```

General form of a member function definition of template class:
```
template <class T>
Ret_type class_name <T>:: function()
{
------------------
------------------
}
```

General form of object creation of a template class:
```
class_name <data_type> object1, object2,.......
```

Program 8.4 **Write a program to add two numbers (either two integers or floats) using class templates.**

Solution:
```
#include <iostream.h>
template <class T>

class Add
{
        T a, b;
        public:
        void getdata();
        void display();
};

template <class T>
void Add <T>::getdata( )
{
        cout<<"Eneter 2 nos";
        cin>>a>>b;
}

template <class T>
void Add <T>::display( )
{
```

```
        cout<<"sum="<<a+b;
}

void main()
{
        Add <int> ob1;
        Add <float> ob2;
        ob1.getdata( );
        ob1.display( );
        ob2.getdata( );
        ob2.display( );
}
```

Output:
Eneter 2 nos   4   5
Sum=9
Eneter 2 nos   4.8   5.1
Sum=9.9

A class with Two Generic Data Types

A template class can have more than one generic data type. Simply declare all the data types required by the class in a comma-separated list within the **template** specification.

Program 8.5   A class with two generic types.

Solution:
```
#include <iostream.h>
template <class Type1, class Type2>
class myclass
{
        Type1 i;
        Type2 j;
        public:
        myclass(Type1 a, Type2 b)
        {
        i = a; j = b;
        }
        void show()
        {
        cout << i << ' ' << j << '\n';
        }
};
```

```
void  main()
{
        myclass<int, double> ob1(10, 0.23);
        myclass<char, char *> ob2('X', "Templates add power.");
        ob1.show(); // show int, double
        ob2.show(); // show char, char *
}
```

The Power of Templates

Templates help you achieve one of the most elusive goals in programming: the creation of reusable code. Through the use of template classes you can create frameworks that can be applied over and over again to a variety of programming situations

Generic functions and classes provide a powerful tool that you can use to amplify your programming efforts. Once you have written and debugged a template class, you have a solid software component that you can use with confidence in a variety of different situations. You are saved from the tedium of creating separate implementations for each data type with which you want the class to work. While it is true that the template syntax can seem a bit intimidating at first, the rewards are well worth the time it takes to become comfortable with it. Template functions and classes are already becoming commonplace in programming, and this trend is expected to continue. For example, the STL (Standard Template Library) defined by C++ is, as its name implies, built upon templates. One last point: although templates add a layer of abstraction, they still ultimately compile down to the same, high-performance object code that you have come to expect from C++.

Short Type Questions
1. Define template.
2. What are generic function and generic class?
3. Write the syntax to define a generic function.
4. Write the syntax to define a generic class.

Long Type Questions
1. "Templates are called parameterized classes or functions". Comment on this line.
2. Write a program in C++ to overload a function template.
3. Write the template function alloc() that takes two parameters:
   n: the size of the array to allocate.
   Val: a value of type T.
   The alloc() function should allocate an array of type T with n elements and set all elements in the array i to value Val , a pointer to array is returned.

Standard Template Library

The **Standard Template Library** (**STL**) is a C++ software library that influenced many parts of the C++ Standard Library It provides four components called algorithms, containers, functional and iterators. The STL provides a ready-made set of common classes for C++, such as containers and associative arrays, that can be used with any built-in type and with any user-defined type that supports some elementary operations (such as copying and assignment). STL algorithms are independent of containers, which significantly reduces the complexity of the library.

The STL achieves its results through the use of templates. This approach provides compile-time polymorphism that is often more efficient than traditional run-time polymorphism. Modern C++ compilers are tuned to minimize any abstraction penalty arising from heavy use of the STL.

At the core of the standard template library are three foundational items: containers, algorithms, and iterators. These items work in conjunction with one another to provide off-the-shelf solutions to a variety of programming problems.

Containers

Containers are objects that hold other objects, and there are several different types. For example, the **vector** class defines a dynamic array, **deque** creates a double-ended queue, and **list** provides a linear list. These containers are called sequence containers because in STL terminology, a sequence is a linear list. In addition to the basic containers, the STL also defines associative containers, which allow efficient retrieval of values based on keys. For example, a **map** provides access to values with unique keys. Thus, a **map** stores a key/value pair and allows a value to be retrieved given its key. Each container class defines a set of functions that may be applied to the container. For example, a list container includes functions that insert, delete, and merge elements. A stack includes functions that push and pop values.

Algorithms

Algorithms act on containers. They provide the means by which you will manipulate the contents of containers. Their capabilities include initialization, sorting, searching, and transforming the contents of containers. Many algorithms operate on a range of elements within a container.

Iterators

Iterators are objects that are, more or less, pointers. They give you the ability to cycle through the contents of a container in much the same way that you would use a pointer to cycle through an array. There are five types of iterators:

Iterator	Access Allowed
Random	Access Store and retrieve values. Elements may be accessed randomly.
Bidirectional	Store and retrieve values. Forward and backward moving.
Forward	Store and retrieve values. Forward moving only.

Input	Retrieve, but not store values. Forward moving only.
Output	Store, but not retrieve values. Forward moving only.

In general, an iterator that has greater access capabilities can be used in place of one that has lesser capabilities. For example, a forward iterator can be used in place of an input iterator. Iterators are handled just like pointers. You can increment and decrement them.

You can apply the * operator to them. Iterators are declared using the **iterator** type defined by the various containers.The STL also supports reverse iterators. Reverse iterators are either bidirectional or random-access iterators that move through a sequence in the reverse direction. Thus, if a reverse iterator points to the end of a sequence, incrementing that iterator will cause it to point to one element before the end.

Other STL Elements

In addition to containers, algorithms, and iterators, the STL relies upon several other standard components for support. Chief among these are allocators, predicates, comparison functions, and function objects. Each container has defined for it an allocator. Allocators manage memory allocation for a container. The default allocator is an object of class **allocator**, but you can define your own allocators if needed by specialized applications. For most uses, the default allocator is sufficient. Several of the algorithms and containers use a special type of function called a predicate. There are two variations of predicates: unary and binary. A unary predicate takes one argument, while a binary predicate has two.

Vectors

Perhaps the most general-purpose of the containers is **vector**. The **vector** class supports a dynamic array. This is an array that can grow as needed. As you know, in C++ the size of an array is fixed at compile time. While this is by far the most efficient way to implement arrays, it is also the most restrictive because the size of the array cannot be adjusted at run time to accommodate changing program conditions. A vector solves this problem by allocating memory as needed. Although a vector is dynamic, you can still use the standard array subscript notation to access its elements.

The template specification for **vector** is shown here:

template <class T, class Allocator = allocator<T>> class vector

Here, **T** is the type of data being stored and **Allocator** specifies the allocator, which defaults to the standard allocator. **vector** has the following constructors:

explicit vector(const Allocator &a = Allocator( ));

explicit vector(size_type num, const T &val = T ( ),

const Allocator &a = Allocator( ));

vector(const vector<T, Allocator> &ob);

template <class InIter> vector(InIter start, InIter end,

const Allocator &a = Allocator( ));

The first form constructs an empty vector. The second form constructs a vector that has num elements with the value val. The value of val may be allowed to default. The third form constructs a vector that contains the same elements as ob. The fourth form constructs a vector that contains the elements in the range specified by the iterators start and end.

Any object that will be stored in a **vector** must define a default constructor. It must also define the < and == operations. Some compilers may require that other comparison operators be defined. (Since implementations vary, consult your compiler's documentation for precise information.) All of the built-in types automatically satisfy these requirements.

Although the template syntax looks rather complex, there is nothing difficult about declaring a vector. Here are some examples:

vector<int> iv; // create zero-length int vector

vector<char> cv(5); // create 5-element char vector

vector<char> cv(5, 'x'); // initialize a 5-element char vector

vector<int> iv2(iv); // create int vector from an int vector

The following comparison operators are defined for **vector**:

==, <, <=, !=, >, >=

The subscripting operator [ ] is also defined for **vector**. This allows you to access the elements of a vector using standard array subscripting notation.

> ### Program 9.1   Basic operation of a vector.

Solution:

```cpp
#include <iostream>
#include <vector>
#include <cctype>
using namespace std;
void main ()
{
        vector<char> v(10); // create a vector of length 10
        int i;
        // display original size of v
        cout << "Size = " << v.size() << endl;
        // assign the elements of the vector some values
        for(i=0; i<10; i++) v[i] = i + 'a';
        // display contents of vector
        cout << "Current Contents:\n";
        for(i=0; i<v.size(); i++) cout << v[i] << " ";
        cout << "\n\n";
        cout << "Expanding vector\n";
        /* put more values onto the end of the vector,
        it will grow as needed */
        for(i=0; i<10; i++) v.push_back(i + 10 + 'a');
```
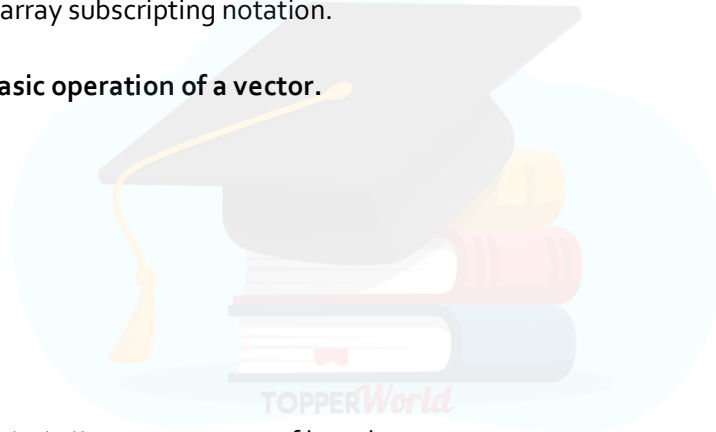
```cpp
// display current size of v
cout << "Size now = " << v.size() << endl;
// display contents of vector
cout << "Current contents:\n";
for(i=o; i<v.size(); i++) cout << v[i] << " ";
cout << "\n\n";
// change contents of vector
for(i=o; i<v.size(); i++) v[i] = toupper(v[i]);
cout << "Modified Contents:\n";
for(i=o; i<v.size(); i++) cout << v[i] << " ";
cout << endl;
}
```

Output:
Size = 10
Current Contents:
a b c d e f g h i j
Expanding vector
Size now = 20
Current contents:
a b c d e f g h i j k l m n o p q r s t
Modified Contents:
A B C D E F G H I J K L M N O P Q R S T

Short Type Questions

1. Differentiate between new and malloc.
2. Differentiate between delete and free.

Long Type Questions

3. Define dynamic memory management. Explain dynamic memory management in C++ with suitable example.

Namespace

The **namespace** keyword allows you to partition the global namespace by creating a declarative region. In essence, a **namespace** defines a scope. The general form of **namespace** is shown here:

namespace name

{

    // declarations

}

Anything defined within a **namespace** statement is within the scope of that namespace.

There is one difference between a class definition and a namespace definition: The namespace is concluded with a closing brace but no terminating semicolon.

Example:

namespace A

{

    int m;

    void display(int n)

    {

    cout<<n;

    }

}

using namespace A;	using namespace A::m;
m=100; //OK	m=100; //OK
display(200); //OK	display(200); // NOT OK, display is not visible

In general, to access a member of a namespace from outside its namespace, precede the member's name with the name of the namespace followed by the scope resolution operator.

Here is a program that demonstrates the use of **CounterNameSpace**.

```
// Demonstrate a namespace.
#include <iostream.h>
namespace CounterNameSpace
{
        int upperbound;
        int lowerbound;
        class counter
        {
                int count;
                public:
                counter(int n)
                {
                if(n <= upperbound) count = n;
                else count = upperbound;
                }
```

```
                void reset(int n)
                {
                if(n <= upperbound) count = n;
                }
                int run()
                {
                if(count > lowerbound) return count--;
                else return lowerbound;
                }
        };
}
void main()
{
        CounterNameSpace::upperbound = 100;
        CounterNameSpace::lowerbound = 0;
        CounterNameSpace::counter ob1(10);
        int i;
        do
        {
        i = ob1.run();
        cout << i << " ";
        } while(i > CounterNameSpace::lowerbound);
        cout << endl;
        CounterNameSpace::counter ob2(20);
        do
        {
        i = ob2.run();
        cout << i << " ";
        } while(i > CounterNameSpace::lowerbound);
        cout << endl;
        ob2.reset(100);
        CounterNameSpace::lowerbound = 90;
        Do
        {
        i = ob2.run();
        cout << i << " ";
        } while(i > CounterNameSpace::lowerbound);
}
```

Notice that the declaration of a **counter** object and the references to **upperbound** and **lowerbound** are qualified by **CounterNameSpace**. However, once an object of type **counter** has been declared, it is not necessary to further qualify it or any of its members. Thus, **ob1.run()** can be called directly; the namespace has already been resolved.

As you can imagine, if your program includes frequent references to the members of a namespace, having to specify the namespace and the scope resolution operator each time you need to refer to one quickly becomes a tedious chore. The **using** statement was invented to alleviate this problem. The **using** statement has these two general forms:

using namespace name;

using name::member;

In the first form, name specifies the name of the namespace you want to access. All of the members defined within the specified namespace are brought into view (i.e., they become part of the current namespace) and may be used without qualification. In the second form, only a specific member of the namespace is made visible. For example, assuming **CounterNameSpace** as shown above, the following **using** statements and assignments are valid.

using CounterNameSpace::lowerbound; // only lowerbound is visible

lowerbound = 10; // OK because lowerbound is visible

using namespace CounterNameSpace; // all members are visible

upperbound = 100; // OK because all members are now visible

Unnamed Namespaces

There is a special type of namespace, called an unnamed namespace that allows you to create identifiers that are unique within a file. Unnamed namespaces are also called anonymous namespaces. They have this general form:

namespace
{
// declarations
}

Unnamed namespaces allow you to establish unique identifiers that are known only within the scope of a single file. That is, within the file that contains the unnamed namespace, the members of that namespace may be used directly, without qualification. But outside the file, the identifiers are unknown. Unnamed namespaces eliminate the need for certain uses of the **static** storage class modifier.

For example, consider the following two files that are part of the same program.

File One

static int k;

void f1() {

k = 99; // OK

}

File Two

extern int k;

void f2() {

k = 10; // error

}

Because k is defined in File One, it may be used in File One. In File Two, k is specified as extern, which means that its name and type are known but that k itself is not actually defined. When these two files are linked, the attempt to use k within File Two results in an error because there is no definition for k. By preceding k with static in File One, its scope is restricted to that file and it is not available to File Two. While the use of static global declarations is still allowed in C++, a better way to accomplish the same effect is to use an unnamed namespace. For example:

File One
```
namespace
{
int k;
}
void f1()
{
k = 99; // OK
}
```

File Two
```
extern int k;
void f2()
{
k = 10; // error
}
```
Here, **k** is also restricted to File One. The use of the unnamed namespace rather than **static** is recommended for new code.

Some Namespace Options

There may be more than one namespace declaration of the same name. This allows a namespace to be split over several files or even separated within the same file.

For example:
```
#include <iostream.h>
namespace NS
{
        int i;
}
namespace NS
{
        int j;
}
void main()
{
        NS::i = NS::j = 10;
        // refer to NS specifically
```

```
        cout << NS::i * NS::j << "\n";
        // use NS namespace
        using namespace NS;
        cout << i * j;
        return o;
}
```

Output:

100

100

Here, NS is split into two pieces. However, the contents of each piece are still within the same namespace, that is, NS.

The std Namespace

Standard C++ defines its entire library in its own namespace called **std**. This is the reason that most of the programs in this book include the following statement:

using namespace std;

This causes the **std** namespace to be brought into the current namespace, which gives you direct access to the names of the functions and classes defined within the library without having to qualify each one with **std::**. Of course, you can explicitly qualify each name with **std::** if you like. For example, the following program does not bring the library into the global namespace.

```
// Use explicit std:: qualification.

#include <iostream>
void main()
{
        int val;
        std::cout << "Enter a number: ";
        std::cin >> val;
        std::cout << "This is your number: ";
        std::cout << std::hex << val;
}
```

Here, **cout, cin,** and the manipulator **hex** are explicitly qualified by their namespace. That is, to write to standard output, you must specify **std::cout**; to read from standard input, you must use **std::cin**; and the hex manipulator must be referred to as **std::hex**.

Creating Conversion Functions

In some situations, you will want to use an object of a class in an expression involving other types of data. Sometimes, overloaded operator functions can provide the means of doing this. However, in other cases, what you want is a simple type conversion from the class type to the target type. To handle these cases, C++ allows you to create custom conversion functions. A conversion function converts your class into a type compatible with that of the rest of the expression. The general format of a type conversion function is:

operator type( ) { return value; }

Here, type is the target type that you are converting your class to, and value is the value of the class after conversion. Conversion functions return data of type type, and no other return type specifier is allowed. Also, no parameters may be included. A conversion function must be a member of the class for which it is defined. Conversion functions are inherited and they may be virtual.

```cpp
#include <iostream.h>
const int SIZE=100;
class stack
{
        int stck[SIZE];
        int tos;
        public:
        stack() { tos=0; }
        void push(int i);
        int pop(void);
        operator int() { return tos; } // conversion of stack to int
};
void stack::push(int i)
{
        if(tos==SIZE)
        {
        cout << "Stack is full.\n";
        return;
        }
        stck[tos] = i;
        tos++;
}
int stack::pop()
{
        if(tos==0)
         {
        cout << "Stack underflow.\n";
        return 0;
        }
```
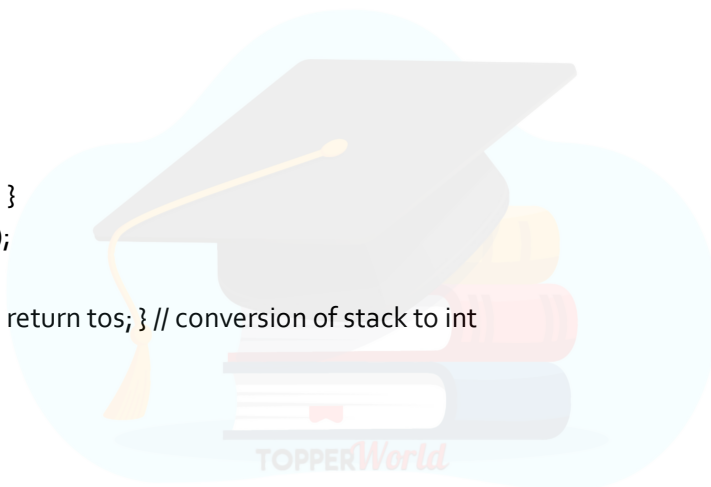
```
        tos--;
        return stck[tos];
}
void main()
{
        stack stck;
        int i, j;
        for(i=0; i<20; i++) stck.push(i);
        j = stck; // convert to integer
        cout << j << " items on stack.\n";
        cout << SIZE - stck << " spaces open.\n";
}
```

Output:
20 items on stack.
80 spaces open.

As the program illustrates, when a stack object is used in an integer expression, such as j = stck, the conversion function is applied to the object. In this specific case, the conversion function returns the value 20. Also, when stck is subtracted from SIZE, the conversion function is also called.