PAVENDAR BHARATHIDASAN COLLEGE OF ARTS AND SCIENCE

(AFFILIATED TO BHARATHIDASAN UNIVERSITY)

BACHELOR OF COMPUTER APPLICATION

PROGRAMMING IN C++

UNIT 1

Basic concepts of Object Oriented Programming – Benefits of OOP – Object Oriented Languages – Applications of OOP – Structure of C++ Program- Tokens , Expressions and Control Structure – Functions in C++.

UNIT 2

Classes and objects – Constructors and Destructors – Operator Overloading and Type Conversions.

UNIT 3

Inheritance: Extending Classes – Pointers – virtual Functions and Polymorphism

UNIT 4

Manging Console I/O Operators – Working with files – Templates – Exception Handling.

UNIT 5

Standard Temple Library – Manipulating Strings – Object Oriented System Development

Prepared By

P.Nagarajan MCA

BCA Dept,

PABCAS

UNIT 1

Basic concepts of Object Oriented Programming – Benefits of OOP – Object Oriented Languages – Applications of OOP – Structure of C++ Program- Tokens , Expressions and Control Structure – Functions in C++.

Object-Oriented Programming

It is a methodology or paradigm to design a program using classes and objects.

The main purpose of using object oriented programming is maintainability and flexibility of the programs.

OOPs concepts used in high level languages such c++ , java, python, .NET.

It simplifies the software development and maintenance by providing some concepts:

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

Object

Object is an Runtime entity or instance of class

Any entity that has **state** and **behavior** is known as an object.

State represents Fields or variables

Behavior represents Functions.

Class

Collection of objects is called class.

Class is also known as blueprint.

It is a logical entity.

Class have State and behaviour , if you want to access the class in main function you should be create object for particular class.

Inheritance

When one object acquires all the properties and behaviours of parent object is known as inheritance.

It provides code reusability.

It is used to achieve runtime polymorphism.

There are 5types of Inheritance

Single Inheritance

Multiple Inheritance

Multilevel Inheritance

Hierarchical Inheritance

Hybrid Inheritance

Extends is the keyword used to achieve Inheritance

Polymorphism

When **one task is performed by different ways** is known as polymorphism.

For example: A person act as a father of his child, son of his father, employee of his company, student of his staff.

Polymorphism is a feature using which an object behaves differently in different situation.

In function overloading we can have more than one function with same name but different numbers, type or sequence of arguments.

In C++, we use two concepts to achieve polymorphism.

Function overloading

Function overriding.

Abstraction

Hiding internal details and showing functionality is known as abstraction.

For example: phone call, we don't know the internal processing.

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

Encapsulation

Binding code and data together into a single unit is known as encapsulation.

For example: capsule, it is wrapped with different medicines.

Encapsulation is also known as Data Hiding

Benefits of OOPs

Reusability of code by using inheritance

To develop and maintain the program easily

Data hiding helps the programmer to build secure programs

It is easy to partition the work in a project based on objects

Object Oriented systems can easily upgraded from small to large systems

Software Complexity can be easily managed

Object Oriented Languages.

The next step in the evolution of computer programming languages, object orientation, was introduced in the **Smalltalk language**.

Object orientation takes the concepts of structured programming one step further. Now, instead of data structures and separate program structures, both data and program elements are combined into one structure called an object.

The object data elements are called attributes, while the object program elements are called methods. Collectively, attributes and methods are called the object's members.

Usually, an object's methods are the only programs able to operate on the object's attributes.

Characteristics of Object-Oriented Programming

Key characteristics of object-oriented programming include

Abstraction

encapsulation

inheritance,

polymorphism.

Some of the Object Oriented Languages are:

Java

C++

Ruby

Python

PHP

Javascript

Swift

Applications of Object Oriented Programming

Main application areas of OOP are:

- User interface design such as windows, menu.
- Real Time Systems
- Simulation and Modeling
- Object oriented databases
- AI and Expert System
- Neural Networks and parallel programming
- Decision support and office automation systems etc.

Structure of C++ Program

C++ program involves the following section:

Documentation

Preprocessor Statements

Global Declarations

The main() function

Local Declarations

Program Statements & Expressions

User Defined Functions

Document Section /* Comments */	Comments are a way of explaining what makes a program. The compiler ignores comments and used by others to understand the code.
Preprocessor statements #include <iostream.h>	This is a *preprocessor directive*. It tells the preprocessor to include the contents of iostream header file in the program before compilation. This file is required for *input-output statements*.
Global Declaration Section int/void	int/void is a return value, which will be explained in a while.
Main Functions main()	The main() is the main function where program execution begins. Every C++ program must contain only one main function.

Example Program

```
/*Hello Program*/

#include <iostream.h>
int main()
{
    cout<<"Hello C++";
    return 0;
}
```

Output Hello C++

C++ Basic Input/Output

C++ I/O operation is using the stream concept. Stream is the sequence of bytes or flow of data. It makes the performance fast.

If bytes flow from main memory to device like printer, display screen, or a network connection, etc, this is called as **output operation.**

If bytes flow from device like printer, display screen, or a network connection, etc to main memory, this is called as **input operation.**

Standard output stream (cout)

The **cout** is a predefined object of **ostream** class.

It is connected with the standard output device, which is usually a display screen.

The cout is used in conjunction with stream insertion operator (<<) to display the output on a console

```
1. #include <iostream>
2. using namespace std;
3. int main( ) {
4.    char ary[] = "Welcome to C++ tutorial";
5.    cout << "Value of ary is: " << ary << endl;
6. }
```

Output: Value of ary is: Welcome to C++ tutorial

Standard input stream (cin)

The **cin** is a predefined object of **istream** class.

It is connected with the standard input device, which is usually a keyboard.

The cin is used in conjunction with stream extraction operator (>>) to read the input from a console.

```
1. #include <iostream>
2. using namespace std;
3. int main( ) {
4.   int age;
5.    cout << "Enter your age: ";
6.    cin >> age;
7.    cout << "Your age is: " << age << endl;
8. }
```

Output:

Enter your age: 22

Your age is: 22

Tokens

Each word and punctuation is referred to as a token in C++.

Tokens are the smallest building block or smallest unit of a C++ program

These following tokens are available in C++:

- Identifiers
- Keywords
- Constants
- Operators
- Strings

C++ Variable or identifiers

A variable is a name of memory location. It is used to store data. Its value can be changed and it can be reused many times.

It is a way to represent memory location through symbol so that it can be easily identified.

int x;
float y;
char z;
	1. **int** x=5,b=10; //declaring 2 variable of integer type
	2. **float** f=30.8;
	3. **char** c='A';

Rules for defining variables

A variable can have alphabets, digits and underscore.

A variable name can start with alphabet and underscore only. It can't start with digit.

No white space is allowed within variable name.

A variable name must not be any reserved word or keyword e.g. char, float etc.

valid variables names	**Invalid variable names**
int a; **int** _ab; **int** a30;	**int** 4; **int** x y; **int double**;

KEYWORDS

Keywords are reserved words which have fixed meaning, and its meaning cannot be changed.

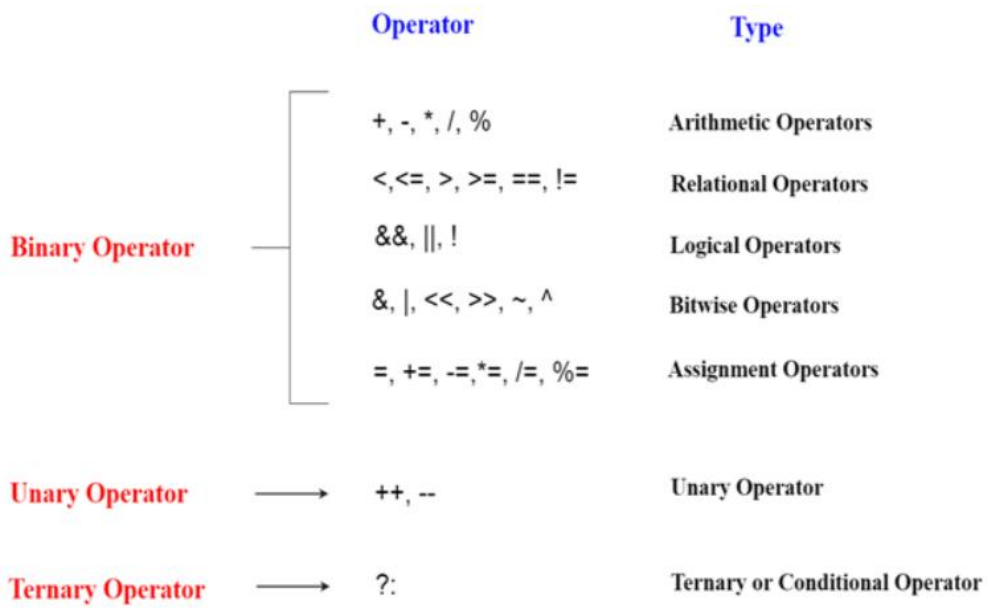
The meaning and working of these keywords are already known to the compiler. C++ has more numbers of keyword than C, and those extra ones have special working capabilities.

C++ Operators

An operator is simply a symbol that is used to perform operations.

There are following types of operators to perform different types of operations in C++ language.

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Assignment Operator
- Unary operator
- Ternary or Conditional Operator



CONSTANTS

Constants are like a variable, except that their value never changes during execution once defined.

STRINGS

Strings are objects that signify sequences of characters

C++ Data Types

A data type specifies the type of data that a variable can store such as integer, floating, character etc.

There are 4 types of data types in C++ language.

Types	Data Types
Basic Data Type	int, char, float, double, etc
Derived Data Type	array, pointer, etc
Enumeration Data Type	Enum
User Defined Data Type	Structure

Basic Data Types

The basic data types are integer-based and floating-point based. C++ language supports both signed and unsigned literals.

The memory size of basic data types may change according to 32 or 64 bit operating system.

Let's see the basic data types. It size is given according to 32 bit OS.

Data Types	Memory Size	Range
Char	1 byte	-128 to 127
signed char	1 byte	-128 to 127
unsigned char	1 byte	0 to 127
Short	2 byte	-32,768 to 32,767
signed short	2 byte	-32,768 to 32,767
unsigned short	2 byte	0 to 32,767
Int	2 byte	-32,768 to 32,767
signed int	2 byte	-32,768 to 32,767
unsigned int	2 byte	0 to 32,767
short int	2 byte	-32,768 to 32,767

signed short int	2 byte	-32,768 to 32,767
unsigned short int	2 byte	0 to 32,767
long int	4 byte	
signed long int	4 byte	
unsigned long int	4 byte	
Float	4 byte	
Double	8 byte	
long double	10 byte	

CONTROL STRUCTURES

In C++ programming, if statement is used to test the condition.

There are various types of if statements in C++.

- if statement
- if-else statement
- nested if statement
- if-else-if ladder

C++ IF Statement

The C++ if statement tests the condition. It is executed if condition is true.

```
1. if(condition){
2. //code to be executed
3. }
```

C++ IF-else Statement

The C++ if-else statement also tests the condition. It executes if block if condition is true otherwise else block is executed.

```
1. if(condition){
2. //code if condition is true
3. }else{
4. //code if condition is false
5. }
```

C++ If-else Example

```
1. #include <iostream>
2. using namespace std;
3. int main () {
4.    int num = 11;
5.          if (num % 2 == 0)
6.          {
7.             cout<<"It is even number";
8.          }
9.          else
10.         {
11.            cout<<"It is odd number";
12.         }
13.   return 0;
14. }
```

OUTPUT: It is odd number

C++ IF-else-if ladder Statement

The C++ if-else-if ladder statement executes one condition from multiple statements.

```
1. if(condition1){
2. //code to be executed if condition1 is true
3. }else if(condition2){
4. //code to be executed if condition2 is true
5. }
6. else if(condition3){
7. //code to be executed if condition3 is true
8. }
9. ...
10. else{
11. //code to be executed if all the conditions are false
12. }
```

C++ If else-if Example

```
1. #include <iostream>
2. using namespace std;
3. int main () {
4.         int num;
5.         cout<<"Enter a number to check grade:";
6.         cin>>num;
7.             if (num <0 || num >100)
8.             {
9.                 cout<<"wrong number";
10.            }
11.            else if(num >= 0 && num < 50){
12.                cout<<"Fail";
13.            }
14.            else if (num >= 50 && num < 60)
15.            {
16.                cout<<"D Grade";
17.            }
18.            else if (num >= 60 && num < 70)
19.            {
20.                cout<<"C Grade";
21.            }
22.            else if (num >= 70 && num < 80)
23.            {
24.                cout<<"B Grade";
25.            }
26.            else if (num >= 80 && num < 90)
27.            {
28.                cout<<"A Grade";
29.            }
30.            else if (num >= 90 && num <= 100)
31.            {
32.                cout<<"A+ Grade";
33.            }
34.    }
```

Output:

```
Enter a number to check grade:66
C Grade
```

Output:

```
Enter a number to check grade:-2
wrong number
```

C++ switch

The C++ switch statement executes one statement from multiple conditions. It is like if-else-if ladder statement in C++.

```
1. switch(expression){
2. case value1:
3.  //code to be executed;
4.  break;
5. case value2:
6.  //code to be executed;
7.  break;
8. ......
9.
10. default:
11.  //code to be executed if all cases are not matched;
12.  break;
13. }
```

C++ Switch Example

```
1. #include <iostream>
2. using namespace std;
3. int main () {
4.         int num;
5.         cout<<"Enter a number to check grade:";
6.         cin>>num;
7.             switch (num)
8.            {
9.                case 10: cout<<"It is 10"; break;
10.               case 20: cout<<"It is 20"; break;
11.               case 30: cout<<"It is 30"; break;
```

```
12.             default: cout<<"Not 10, 20 or 30"; break;
13.         }
14.     }
```

Output:

```
Enter a number:
10
It is 10
```

C++ For Loop

The C++ for loop is used to iterate a part of the program several times.

If the number of iteration is fixed, it is recommended to use for loop than while or do-while loops.

The C++ for loop is same as C/C#. We can initialize variable, check condition and increment/decrement value.

```
1. for(initialization; condition; incr/decr){
2. //code to be executed
3. }
```

C++ For Loop Example

```
1. #include <iostream>
2. using namespace std;
3. int main() {
4.         for(int i=1;i<=10;i++){
5.             cout<<i <<"\n";
6.         }
7.     }
```

Output:

```
1
2
3
4
5
6
7
8
9
10
```

C++ Nested For Loop Example

Let's see a simple example of nested for loop in C++.

```
#include <iostream>
using namespace std;
int main () {
        for(int i=1;i<=3;i++){
            for(int j=1;j<=3;j++){
            cout<<i<<" "<<j<<"\n";
          }
        }
    }
```

Output:

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

C++ While loop

In C++, while loop is used to iterate a part of the program several times.

If the number of iteration is not fixed, it is recommended to use while loop than for loop.

```
while(condition){
//code to be executed
}
```

```
#include <iostream>
using namespace std;
int main() {
 int i=1;
        while(i<=10)
       {
            cout<<i <<"\n";
            i++;
          }
    }
```

Output:

```
1
2
3
4
5
6
7
8
9
10
```

C++ Do-While Loop

The C++ do-while loop is used to iterate a part of the program several times.

If the number of iteration is not fixed and you must have to execute the loop at least once, it is recommended to use do-while loop.

The C++ do-while loop is executed at least once because condition is checked after loop body.

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

C++ do-while Loop Example

Let's see a simple example of C++ do-while loop to print the table of 1.

```
#include <iostream>
using namespace std;
int main() {
    int i = 1;
        do{
            cout<<i<<"\n";
            i++;
        } while (i <= 10) ;
}
```

C++ Break Statement

The C++ break is used to break loop or switch statement. It breaks the current flow of the program at the given condition.

In case of inner loop, it breaks only inner loop.

1. jump-statement;
2. **break**;

C++ Break Statement Example

Let's see a simple example of C++ break statement which is used inside the loop

```
1. #include <iostream>
2. using namespace std;
3. int main() {
4.       for (int i = 1; i <= 10; i++)
5.          {
6.             if (i == 5)
7.             {
8.                break;
9.             }
10.        cout<<i<<"\n";
11.         }
12. }
```

Output:

```
1
2
3
4
```

C++ Continue Statement

The C++ continue statement is used to continue loop.

It continues the current flow of the program and skips the remaining code at specified condition.

In case of inner loop, it continues only inner loop.

1. jump-statement;
2. **continue**;

C++ Continue Statement Example

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

```
5.      for(int i=1;i<=10;i++){
6.          if(i==5){
7.              continue;
8.          }
9.          cout<<i<<"\n";
10.     }
11. }
```

Output:

```
1
2
3
4
6
7
8
9
10
```

C++ Goto Statement

The C++ goto statement is also known as jump statement.

It is used to transfer control to the other part of the program.

It unconditionally jumps to the specified label.

It can be used to transfer control from deeply nested loop or switch case label.

C++ Goto Statement Example

Let's see the simple example of goto statement in C++.

```
1.  #include <iostream>
2.  using namespace std;
3.  int main()
4.  {
5.  ineligible:
6.        cout<<"You are not eligible to vote!\n";
7.      cout<<"Enter your age:\n";
8.      int age;
9.      cin>>age;
10.     if (age < 18){
11.         goto ineligible;
12.     }
13.     else
14.     {
15.         cout<<"You are eligible to vote!";
16.     }
17. }
```

Output:

```
You are not eligible to vote!
Enter your age:
16
You are not eligible to vote!
Enter your age:
7
You are not eligible to vote!
Enter your age:
22
You are eligible to vote!
```

C++ Functions

The function in C++ language is also known as procedure or subroutine in other programming languages.

To perform any task, we can create function.

A function can be called many times.

It provides modularity and code reusability.

Advantage of functions in C

There are many advantages of functions.

1) Code Reusability

By creating functions in C++, you can call it many times.

So we don't need to write the same code again and again.

2) Code optimization

It makes the code optimized, we don't need to write much code.

Suppose, you have to check 3 numbers (531, 883 and 781) whether it is prime number or not. Without using function, you need to write the prime number logic 3 times.

So, there is repetition of code.

But if you use functions, you need to write the logic only once and you can reuse it several times.

Types of Functions

There are two types of functions in C programming:

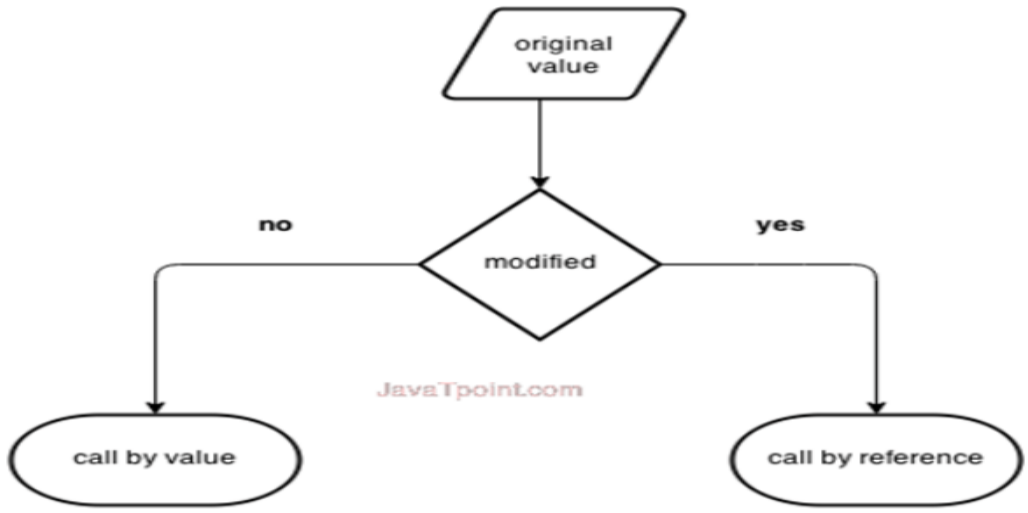
1. Library Functions: are the functions which are declared in the C++ header files such as ceil(x), cos(x), exp(x), etc.

2. User-defined functions: are the functions which are created by the C++ programmer, so that he/she can use it many times. It reduces complexity of a big program and optimizes the code.

Call by value and call by reference in C++

There are two ways to pass value or data to function in C language: call by value and call by reference.

Original value is not modified in call by value but it is modified in call by reference.



Let's understand call by value and call by reference in C++ language one by one.

Call by value in C++

In call by value, **original value is not modified.**

```
1. #include <iostream>
2. using namespace std;
3. void change(int data);
4. int main()
5. {
6. int data = 3;
7. change(data);
8. cout << "Value of the data is: " << data<< endl;
9. return 0;
10. }
11. void change(int data)
12. {
13. data = 5;
14. }
```

Output:

```
Value of the data is: 3
```

Call by reference in C++

In call by reference, original value is modified because we pass reference (address).

Note: To understand the call by reference, you must have the basic knowledge of pointers.

Let's try to understand the concept of call by reference in C++ language by the example given below:

```
1. #include<iostream>
2. using namespace std;
3. void swap(int *x, int *y)
4. {
5.  int swap;
6.  swap=*x;
7.  *x=*y;
8.  *y=swap;
9. }
10. int main()
11. {
12.  int x=500, y=100;
13.  swap(&x, &y);  // passing value to function
14.  cout<<"Value of x is: "<<x<<endl;
15.  cout<<"Value of y is: "<<y<<endl;
16.  return 0;
17. }
```

Output:

```
Value of x is: 100
Value of y is: 500
```

C++ Recursion

When function is called within the same function, it is known as recursion in C++.

The function which calls the same function, is known as recursive function.

A function that calls itself, and doesn't perform any task after function call, is known as tail recursion.

In tail recursion, we generally call the same function with return statement.

```
1. recursionfunction()
2. {
3. recursionfunction(); //calling self function
4. }
```

UNIT 2

Classes and objects – Constructors and Destructors – Operator Overloading and Type Conversions.

Class and objects program

C++ Class Example: Store and Display Employee Information

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

```
1. #include <iostream>
2. using namespace std;
3. class Employee {
4.   public:
5.      int id;//data member (also instance variable)
6.      string name;//data member(also instance variable)
7.      float salary;
8.      void insert(int i, string n, float s)
9.       {
10.        id = i;
11.        name = n;
12.        salary = s;
13.      }
14.     void display()
15.      {
16.        cout<<id<<"  "<<name<<"  "<<salary<<endl;
17.      }
18. };
19. int main(void) {
20.    Employee e1; //creating an object of Employee
21.    Employee e2; //creating an object of Employee
22.    e1.insert(201, "Sonoo",990000);
23.    e2.insert(202, "Nakul", 29000);
24.    e1.display();
25.    e2.display();
26.    return 0;
27. }
```

Output:

```
201  Sonoo  990000
202  Nakul  29000
```

CONSTRUCTORS

It is used to initialize the value to the variable in class.

Constructor name function name should be same

Constructor should not have return type.

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

- Default constructor
- Parameterized constructor

C++ Default Constructor

A constructor which has no argument is known as default constructor.

It is invoked at the time of creating object.

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

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

Output:

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

C++ Parameterized Constructor

A constructor which has parameters is called parameterized constructor.

It is used to provide different values to distinct objects.

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

```
#include <iostream>
using namespace std;
class Employee {
  public:
     int id;//data member (also instance variable)
     string name;//data member(also instance variable)
```

```
    float salary;
    Employee(int i, string n, float s)
     {
        id = i;
        name = n;
        salary = s;
     }
    void display()
     {
        cout<<id<<"  "<<name<<"  "<<salary<<endl;
     }
};
int main(void) {
   Employee e1 =Employee(101, "Sonoo", 890000); //creating an object of Employee
   Employee e2=Employee(102, "Nakul", 59000);
   e1.display();
   e2.display();
   return 0;
}
```

Output:

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

C++ Copy Constructor

A Copy constructor is an **overloaded** constructor used to declare and initialize an object from another object

C++ Destructor

A destructor works opposite to constructor; it destructs the objects of classes.

It can be defined only once in a class.

Like constructors, it is invoked automatically.

A destructor is defined like constructor.

It must have same name as class.

But it is prefixed with a tilde sign (~).

C++ Constructor and Destructor Example

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

```
1. #include <iostream>
2. using namespace std;
3. class Employee
4.  {
5.    public:
6.        Employee()
7.        {
8.            cout<<"Constructor Invoked"<<endl;
```

```
9.          }
10.         ~Employee()
11.         {
12.            cout<<"Destructor Invoked"<<endl;
13.         }
14. };
15. int main(void)
16. {
17.    Employee e1; //creating an object of Employee
18.    Employee e2; //creating an object of Employee
19.    return 0;
20. }
```

Output:

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

Types of overloading in C++ are:

- Function overloading
- Operator overloading

C++ Function Overloading

Function Overloading is defined as the process of having two or more function with the same name, but different in parameters is known as function overloading in C++

C++ Operators Overloading

Operator overloading is a compile-time polymorphism in which the operator is overloaded to provide the special meaning to the user-defined data type.

Operator overloading is used to overload or redefines most of the operators available in C++. It is used to perform the operation on the user-defined data type.

For example, C++ provides the ability to add the variables of the user-defined data type that is applied to the built-in data types.

The advantage of Operators overloading is to perform different operations on the same operand.

Operator that cannot be overloaded are as follows:

- Scope operator (::)
- Sizeof
- member selector(.)
- member pointer selector(*)
- ternary operator(?:)

Syntax of Operator Overloading

```
return_type class_name  : : operator op(argument_list)
{
    // body of the function.
}
```

Where the **return type** is the type of value returned by the function.

class_name is the name of the class.

operator op is an operator function where op is the operator being overloaded, and the operator is the keyword.

Rules for Operator Overloading

- Existing operators can only be overloaded, but the new operators cannot be overloaded.
- The overloaded operator contains atleast one operand of the user-defined data type.
- We cannot use friend function to overload certain operators. However, the member function can be used to overload those operators.
- When unary operators are overloaded through a member function take no explicit arguments, but, if they are overloaded by a friend function, takes one argument.
- When binary operators are overloaded through a member function takes one explicit argument, and if they are overloaded through a friend function takes two explicit arguments.

C++ Operators Overloading Example

Let's see the simple example of operator overloading in C++. In this example, void operator ++ () operator function is defined (inside Test class).

// program to overload the unary operator ++.

```
#include <iostream>
using namespace std;
class Test
{
  private:
    int num;
  public:
     Test(): num(8){}
     void operator ++()        {
       num = num+2;
     }
     void Print() {
        cout<<"The Count is: "<<num;
     }
```

```
};
int main()
{
    Test tt;
    ++tt;  // calling of a function "void operator ++()"
    tt.Print();
    return 0;
}
```

Output:

```
The Count is: 10
```

Type Conversion in C++

A type cast is basically a conversion from one type to another.

There are two types of type conversion:

1. **Implicit Type Conversion**

   Also known as 'automatic type conversion'.

- Done by the compiler on its own, without any external trigger from the user.
- Generally takes place when in an expression more than one data type is present. In such condition type conversion (type promotion) takes place to avoid lose of data.
- All the data types of the variables are upgraded to the data type of the variable with largest data type.

```
bool -> char -> short int -> int ->

unsigned int -> long -> unsigned ->

long long -> float -> double -> long double
```

Explicit Type Conversion:

This process is also called type casting and it is user-defined.

Here the user can typecast the result to make it of a particular data type.

Advantages of Type Conversion:

- This is done to take advantage of certain features of type hierarchies or type representations.
- It helps to compute expressions containing variables of different data types.

UNIT 3

Inheritance: Extending Classes – Pointers – virtual Functions and Polymorphism

C++ Inheritance

In C++, inheritance is a process in which one object acquires all the properties and behaviors of its parent object automatically.

In such way, you can reuse, extend or modify the attributes and behaviors which are defined in other class.

Advantage of C++ Inheritance

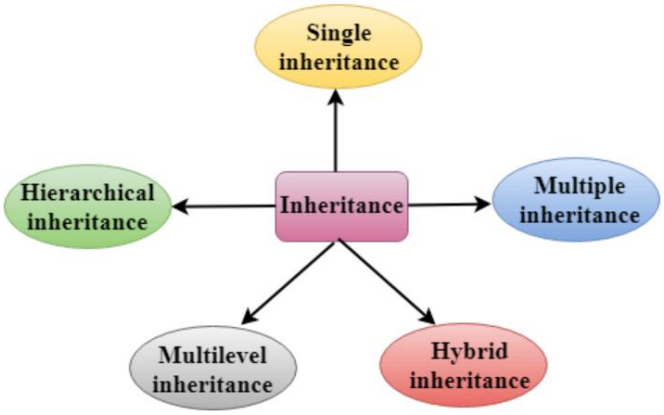
Code reusability

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

Types Of Inheritance

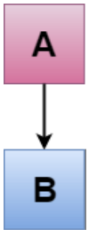
C++ supports five types of inheritance:

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



C++ Single Inheritance

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



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

C++ Single Level Inheritance Example: Inheriting Fields

When one class inherits another class, it is known as single level inheritance.

Let's see the example of single level inheritance which inherits the fields only.

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

Output

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

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

C++ Single Level Inheritance Example: Inheriting Methods

Let's see another example of inheritance in C++ which inherits methods only.

```
1. #include <iostream>
2. using namespace std;
3.  class Animal {
4.    public:
5.  void eat() {
6.     cout<<"Eating..."<<endl;
7.  }
8.    };
9.    class Dog: public Animal
```

```
10.   {
11.       public:
12.     void bark(){
13.    cout<<"Barking...";
14.     }
15.   };
16. int main(void) {
17.    Dog d1;
18.    d1.eat();
19.    d1.bark();
20.    return 0;
21. }
```

Output:

```
Eating...
Barking...
```

Let's see a simple example.

```
1.  #include <iostream>
2.  using namespace std;
3.  class A
4.  {
5.     int a = 4;
6.     int b = 5;
7.     public:
8.     int mul()
9.     {
10.       int c = a*b;
11.       return c;
12.    }
13. };
14.
15. class B : private A
16. {
17.    public:
18.    void display()
19.    {
20.       int result = mul();
21.       std::cout <<"Multiplication of a and b is : "<<result<< std::endl;
22.    }
23. };
24. int main()
25. {
26.   B b;
27.   b.display();
28.    return 0;
29. }
```

Output:

```
Multiplication of a and b is : 20
```

In the above example, class A is privately inherited.

Therefore, the mul() function of class 'A' cannot be accessed by the object of class B.

It can only be accessed by the member function of class B.

How to make a Private Member Inheritable

The private member is not inheritable.

If we modify the visibility mode by making it public, but this takes away the advantage of data hiding.

C++ introduces a third visibility modifier, i.e., **protected**.

The member which is declared as protected will be accessible to all the member functions within the class as well as the class immediately derived from it.

Visibility modes can be classified into three categories:

- **Public**: When the member is declared as public, it is accessible to all the functions of the program.
- **Private**: When the member is declared as private, it is accessible within the class only.
- **Protected**: When the member is declared as protected, it is accessible within its own class as well as the class immediately derived from it.

C++ Multilevel Inheritance

Multilevel inheritance is a process of deriving a class from another derived class.

C++ Multi Level Inheritance Example

When one class inherits another class which is further inherited by another class, it is known as multi level inheritance in C++

Let's see the example of multi level inheritance in C++.

```
1. #include <iostream>
2. using namespace std;
3.  class Animal {
4.    public:
5.  void eat() {
```

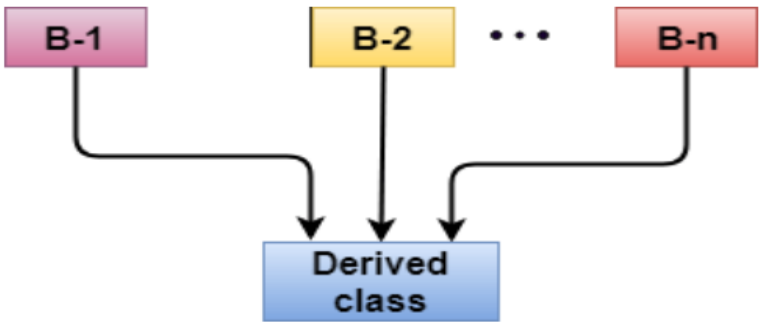
```
6.      cout<<"Eating..."<<endl;
7.    }
8.     };
9.     class Dog: public Animal
10.   {
11.      public:
12.     void bark(){
13.    cout<<"Barking..."<<endl;
14.     }
15.   };
16.   class BabyDog: public Dog
17.   {
18.      public:
19.     void weep() {
20.    cout<<"Weeping...";
21.     }
22.   };
23. int main(void) {
24.    BabyDog d1;
25.    d1.eat();
26.    d1.bark();
27.     d1.weep();
28.     return 0;
29. }
```

Output:

```
Eating...
Barking...
Weeping...
```

C++ Multiple Inheritance

Multiple inheritance is the process of deriving a new class that inherits the attributes from two or more classes.



Let's see a simple example of multiple inheritance.

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

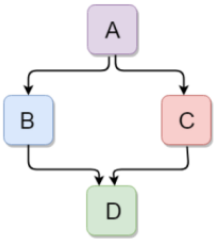
```
3.  class A
4.  {
5.      protected:
6.       int a;
7.      public:
8.      void get_a(int n)
9.      {
10.         a = n;
11.     }
12. };
13.
14. class B
15. {
16.     protected:
17.     int b;
18.     public:
19.     void get_b(int n)
20.     {
21.         b = n;
22.     }
23. };
24. class C : public A,public B
25. {
26.    public:
27.     void display()
28.     {
29.         std::cout << "The value of a is : " <<a<< std::endl;
30.         std::cout << "The value of b is : " <<b<< std::endl;
31.         cout<<"Addition of a and b is : "<<a+b;
32.     }
33. };
34. int main()
35. {
36.    C c;
37.    c.get_a(10);
38.    c.get_b(20);
39.    c.display();
40.
41.     return 0;
42. }
```

Output:

```
The value of a is : 10
The value of b is : 20
Addition of a and b is : 30
```

C++ Hybrid Inheritance

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



Let's see a simple example:

```
#include <iostream>
using namespace std;
class A
{
   protected:
   int a;
   public:
   void get_a()
   {
      std::cout << "Enter the value of 'a' : " << std::endl;
      cin>>a;
   }
};

class B : public A
{
   protected:
   int b;
   public:
   void get_b()
   {
       std::cout << "Enter the value of 'b' : " << std::endl;
      cin>>b;
   }
};
class C
{
   protected:
   int c;
   public:
   void get_c()
   {
      std::cout << "Enter the value of c is : " << std::endl;
      cin>>c;
```

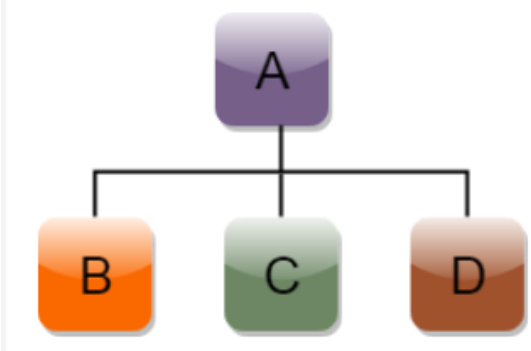
```
35.     }
36. };
37.
38. class D : public B, public C
39. {
40.     protected:
41.     int d;
42.     public:
43.     void mul()
44.     {
45.         get_a();
46.         get_b();
47.         get_c();
48.         std::cout << "Multiplication of a,b,c is : " <<a*b*c<< std::endl;
49.     }
50. };
51. int main()
52. {
53.     D d;
54.     d.mul();
55.     return 0;
56. }
```

Output:

```
Enter the value of 'a' :
10
Enter the value of 'b' :
20
Enter the value of c is :
30
Multiplication of a,b,c is : 6000
```

C++ Hierarchical Inheritance

Hierarchical inheritance is defined as the process of deriving more than one class from a base class.



Let's see a simple example:

```
1. #include <iostream>
2. using namespace std;
3. class Shape                // Declaration of base class.
4. {
5.     public:
6.     int a;
7.     int b;
8.     void get_data(int n,int m)
9.     {
10.        a= n;
11.        b = m;
12.    }
13. };
14. class Rectangle : public Shape  // inheriting Shape class
15. {
16.    public:
17.    int rect_area()
18.    {
19.        int result = a*b;
20.        return result;
21.    }
22. };
23. class Triangle : public Shape    // inheriting Shape class
24. {
25.    public:
26.    int triangle_area()
27.    {
28.        float result = 0.5*a*b;
29.        return result;
30.    }
31. };
32. int main()
33. {
34.    Rectangle r;
35.    Triangle t;
36.    int length,breadth,base,height;
37.    std::cout << "Enter the length and breadth of a rectangle: " << std::endl;
38.    cin>>length>>breadth;
39.    r.get_data(length,breadth);
40.    int m = r.rect_area();
41.    std::cout << "Area of the rectangle is : " <<m<< std::endl;
42.    std::cout << "Enter the base and height of the triangle: " << std::endl;
43.    cin>>base>>height;
```

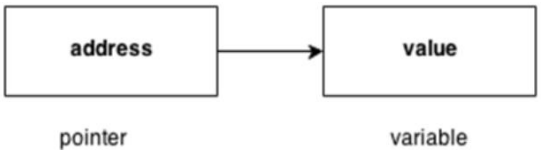
```
44.    t.get_data(base,height);
45.    float n = t.triangle_area();
46.    std::cout <<"Area of the triangle is : "  << n<<std::endl;
47.    return 0;
48. }
```

Output:

```
Enter the length and breadth of a rectangle:
23
20
Area of the rectangle is : 460
Enter the base and height of the triangle:
2
5
Area of the triangle is : 5
```

C++ Pointers

The pointer in C++ language is a variable, it is also known as locator or indicator that points to an address of a value.



Advantage of pointer

1) Pointer reduces the code and improves the performance, it is used to retrieving strings, trees etc. and used with arrays, structures and functions.

2) We can return multiple values from function using pointer.

3) It makes you able to access any memory location in the computer's memory.

Usage of pointer

There are many usage of pointers in C++ language.

1) Dynamic memory allocation

In c language, we can dynamically allocate memory using malloc() and calloc() functions where pointer is used.

2) Arrays, Functions and Structures

Pointers in c language are widely used in arrays, functions and structures. It reduces the code and improves the performance.

Symbols used in pointer

Symbol	Name	Description
& (ampersand sign)	Address operator	Determine the address of a variable.
* (asterisk sign)	Indirection operator	Access the value of an address.

Pointer Example

Let's see the simple example of using pointers printing the address and value.

```
1. #include <iostream>
2. using namespace std;
3. int main()
4. {
5. int number=30;
6. int * p;
7. p=&number;//stores the address of number variable
8. cout<<"Address of number variable is:"<<&number<<endl;
9. cout<<"Address of p variable is:"<<p<<endl;
10. cout<<"Value of p variable is:"<<*p<<endl;
11.  return 0;
12. }
```

Output

```
Address of number variable is:0x7ffccc8724c4
Address of p variable is:0x7ffccc8724c4
Value of p variable is:30
```

virtual function

A virtual function is a special form of member function that is declared within a base class and redefined by a derived class.

The keyword virtual is used to create a virtual function, precede the function's declaration in the base class.

If a class includes a virtual function and if it gets inherited, the virtual class redefines a virtual function to go with its own need.

In other words, a virtual function is a function which gets override in the derived class and instructs the C++ compiler for executing late binding on that function.

A function call is resolved at runtime in late binding and so compiler determines the type of object at runtime.

```
#include<iostream.h>
class b
{
public:
virtual void show()
{
cout<<"\n  Showing base class....";
}
void display()
{
cout<<"\n  Displaying base class...." ;
}
};

class d:public b
{
public:
void display()
{
cout<<"\n  Displaying derived class....";
}
void show()
{
cout<<"\n  Showing derived class....";
}
};

int main()
{
b B;
b *ptr;
cout<<"\n\t P points to base:\n" ; ptr=&B; ptr->display();
ptr->show();
cout<<"\n\n\t P points to drive:\n"; d D; ptr=&D; ptr->display();
ptr->show();
}
```

Program Output:

```
    P points to base:

Displaying base class....
Showing base class....

    P points to drive:
```

```
Displaying base class....
Showing derived class....
```

C++ Polymorphism

The term "Polymorphism" is the combination of "poly" + "morphs" which means many forms.

It is a greek word. In object-oriented programming, we use 3 main concepts: inheritance, encapsulation, and polymorphism.

Real Life Example Of Polymorphism

Let's consider a real-life example of polymorphism.

A lady behaves like a teacher in a classroom, mother or daughter in a home and customer in a market. Here, a single person is behaving differently according to the situations.

There are two types of polymorphism in C++:

- **Compile time polymorphism**

  The overloaded functions are invoked by matching the type and number of arguments.

  This information is available at the compile time and, therefore, compiler selects the appropriate function at the compile time.

  It is achieved by function overloading and operator overloading which is also known as static binding or early binding.

  Now, let's consider the case where function name and prototype is same.

```
1.  class A                          //  base class declaration.
2.  {
3.      int a;
4.      public:
5.      void display()
6.      {
7.          cout<< "Class A ";
8.      }
9.  };
10. class B : public A                 //  derived class declaration.
11. {
12.   int b;
13.   public:
14.  void display()
15.  {
16.      cout<<"Class B";
17.  }
18. };
```

In the above case, the prototype of display() function is the same in both the **base and derived class**.

Therefore, the static binding cannot be applied.

It would be great if the appropriate function is selected at the run time. This is known as **run time polymorphism**.

- **Run time polymorphism**

  Run time polymorphism is achieved when the object's method is invoked at the run time instead of compile time.

  It is achieved by method overriding which is also known as dynamic binding or late binding.

Differences b/w compile time and run time polymorphism.

Compile time polymorphism	Run time polymorphism
The function to be invoked is known at the compile time.	The function to be invoked is known at the run time.
It is also known as overloading, early binding and static binding.	It is also known as overriding, Dynamic binding and late binding.
Overloading is a compile time polymorphism where more than one method is having the same name but with the different number of parameters or the type of the parameters.	Overriding is a run time polymorphism where more than one method is having the same name, number of parameters and the type of the parameters.
It is achieved by function overloading and operator overloading.	It is achieved by virtual functions and pointers.
It provides fast execution as it is known at the compile time.	It provides slow execution as it is known at the run time.
It is less flexible as mainly all the things execute at the compile time.	It is more flexible as all the things execute at the run time.

C++ Runtime Polymorphism Example

Let's see a simple example of run time polymorphism in C++.

```
// an example without the virtual keyword.
```

```
1. #include <iostream>
2. using namespace std;
3. class Animal {
4.     public:
5. void eat(){
6. cout<<"Eating...";
7.     }
8. };
9. class Dog: public Animal
10. {
11.  public:
12.  void eat()
13.    {          cout<<"Eating bread...";
14.    }
15. };
16. int main(void) {
17.   Dog d = Dog();
18.   d.eat();
19.   return 0;
20. }
```

Output:

```
Eating bread...
```

C++ Run time Polymorphism Example: By using two derived class

Let's see another example of run time polymorphism in C++ where we are having two derived classes.

```
// an example with virtual keyword.
1. #include <iostream>
2. using namespace std;
3. class Shape {                          //  base class
4.     public:
5. virtual void draw(){                   // virtual function
6. cout<<"drawing..."<<endl;
7.     }
8. };
9. class Rectangle: public Shape              //  inheriting Shape class.
10. {
11.  public:
12.  void draw()
13.    {
14.       cout<<"drawing rectangle..."<<endl;
15.    }
16. };
17. class Circle: public Shape                  //  inheriting Shape class.
```

```
18.
19. {
20.  public:
21.  void draw()
22.    {
23.       cout<<"drawing circle..."<<endl;
24.    }
25. };
26. int main(void) {
27.     Shape *s;                    //  base class pointer.
28.     Shape sh;                    // base class object.
29.        Rectangle rec;
30.         Circle cir;
31.       s=&sh;
32.      s->draw();
33.         s=&rec;
34.      s->draw();
35.     s=?
36.     s->draw();
37. }
```

Output:

```
drawing...
drawing rectangle...
drawing circle...
```

Runtime Polymorphism with Data Members

Runtime Polymorphism can be achieved by data members in C++.

Let's see an example where we are accessing the field by reference variable which refers to the instance of derived class.

```
1.  #include <iostream>
2.  using namespace std;
3.  class Animal {                          //  base class declaration.
4.     public:
5.     string color = "Black";
6.  };
7.  class Dog: public Animal                 // inheriting Animal class.
8.  {
9.   public:
10.    string color = "Grey";
11. };
12. int main(void) {
13.     Animal d= Dog();
14.    cout<<d.color;
15. }
```

Output: Black

UNIT 4

Manging Console I/O Operators – Working with files – Templates – Exception Handling.

C++ Stream Classes

What is Stream?

- A stream is an abstraction. It is a sequence of bytes.
- It represents a device on which input and output operations are performed.
- It can be represented as a source or destination of characters of indefinite length.
- It is generally associated to a physical source or destination of characters like a disk file, keyboard or console.
- C++ provides standard **iostream** library to operate with streams.
- The **iostream** is an object-oriented library which provides Input/Output functionality using streams.

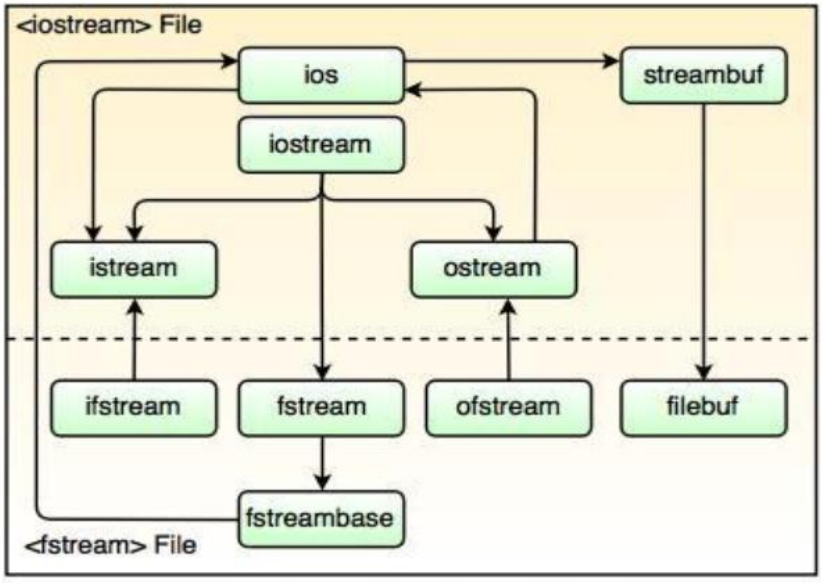


Fig. Stream Class Hierarchy

I/O Stream	Meaning	Description
istream	Input Stream	It reads and interprets input.
ostream	Output stream	It can write sequences of characters and represents other kinds of data.
ifstream	Input File Stream	The ifstream class is derived from fstreambase and istream by multiple inheritance. This class accesses the member functions such as get(), getline(),

		seekg(), tellg() and read(). It provides open() function with the default input mode and allows input operations.
ofstream	Output File Stream	The ofstream class is derived from fstreambase and ostream classes. This class accesses the member functions such as put(), seekp(), write() and tellp(). It provides the member function open() with the default output mode.
fstream	File Stream	The fstream allows input and output operations simultaneous on a filebuf. It invokes the member function istream::getline() to read characters from the file. This class provides the open() function with the default input mode.
fstreambase	File Stream Base	It acts as a base class for fstream, ifstream and ofstream. The open() and close() functions are defined in fstreambase.

Advantages of Stream Classes

- Stream classes have good error handling capabilities.
- These classes work as an abstraction for the user that means the internal operation is encapsulated from the user.
- These classes are buffered and do not uses the memory disk space.
- These classes have various functions that make reading or writing a sequence of bytes easy for the programmer.

File Handling using File Streams in C++

File represents storage medium for storing data or information. Streams refer to sequence of bytes.

In Files we store data i.e. text or binary data permanently and use these data to read or write in the form of input output operations by transferring bytes of data.

So we use the term File Streams/File handling. We use the header file <fstream>

- **ofstream:** It represents output Stream and this is used for writing in files.
- **ifstream:** It represents input Stream and this is used for reading from files.
- **fstream:** It represents both output Stream and input Stream. So it can read from files and write to files.

Operations in File Handling:

- Creating a file: open()
- Reading data: read()
- Writing new data: write()
- Closing a file: close()

Creating/Opening a File

We create/open a file by specifying new path of the file and mode of operation. Operations can be reading, writing, appending and truncating.

Syntax for file creation: FilePointer.open("Path",ios::mode);

- Example of file opened for writing: st.open("E:\studytonight.txt",ios::out);
- Example of file opened for reading: st.open("E:\studytonight.txt",ios::in);
- Example of file opened for appending: st.open("E:\studytonight.txt",ios::app);
- Example of file opened for truncating: st.open("E:\studytonight.txt",ios::trunc);

CREATING A FILE

```
#include<iostream>
#include<conio>
#include <fstream>

using namespace std;

int main()
{
    fstream st; // Step 1: Creating object of fstream class
    st.open("E:\studytonight.txt",ios::out);  // Step 2: Creating new file
    if(!st) // Step 3: Checking whether file exist
    {
        cout<<"File creation failed";
    }
    else
    {
        cout<<"New file created";
        st.close(); // Step 4: Closing file
    }
    getch();
    return 0;
}
```

Writing to a File

```
#include <iostream>
#include<conio>
#include <fstream>

using namespace std;

int main()
{
    fstream st; // Step 1: Creating object of fstream class
    st.open("E:\studytonight.txt",ios::out);  // Step 2: Creating new file
    if(!st) // Step 3: Checking whether file exist
    {
        cout<<"File creation failed";
    }
    else
    {
        cout<<"New file created";
        st<<"Hello";    // Step 4: Writing to file
        st.close(); // Step 5: Closing file
    }
    getch();
    return 0;
}
```

Here we are sending output to a file. So, we use ios::out. As given in the program, information typed inside the quotes after **"FilePointer <<"** will be passed to output file.

C++ Exception Handling

Exception Handling in C++ is a process to handle runtime errors.

We perform exception handling so the normal flow of the application can be maintained even after runtime errors.

In C++, exception is an event or object which is thrown at runtime. All exceptions are derived from std::exception class.

It is a runtime error which can be handled. If we don't handle the exception, it prints exception message and terminates the program.

Advantage

It maintains the normal flow of the application. In such case, rest of the code is executed even after exception.

C++ Exception Classes

In C++ standard exceptions are defined in <exception> class that we can use inside our programs. The arrangement of parent-child class hierarchy is shown below:

All the exception classes in C++ are derived from std::exception class.

Let's see the list of C++ common exception classes.

Exception	Description
std::exception	It is an exception and parent class of all standard C++ exceptions.
std::logic_failure	It is an exception that can be detected by reading a code.
std::runtime_error	It is an exception that cannot be detected by reading a code.
std::bad_exception	It is used to handle the unexpected exceptions in a c++ program.
std::bad_cast	This exception is generally be thrown by **dynamic_cast.**
std::bad_typeid	This exception is generally be thrown by **typeid.**
std::bad_alloc	This exception is generally be thrown by **new.**

C++ Exception Handling Keywords

In C++, we use 3 keywords to perform exception handling:

- try
- catch, and
- throw

Moreover, we can create user-defined exception which we will learn in next chapters.

C++ try/catch

In C++ programming, exception handling is performed using try/catch statement.

The C++ **try block** is used to place the code that may occur exception.

The **catch block** is used to handle the exception.

C++ example without try/catch

```
1. #include <iostream>
2. using namespace std;
3. float division(int x, int y) {
4.    return (x/y);
5. }
6. int main () {
7.    int i = 50;
8.    int j = 0;
9.    float k = 0;
10.      k = division(i, j);
11.      cout << k << endl;
12.   return 0;
13. }
```

Output:

```
Floating point exception (core dumped)
```

C++ try/catch example

```
1. #include <iostream>
2. using namespace std;
3. float division(int x, int y) {
4.   if( y == 0 ) {
5.     throw "Attempted to divide by zero!";
6.   }
7.   return (x/y);
8. }
9. int main () {
10.  int i = 25;
11.  int j = 0;
12.  float k = 0;
13.  try {
14.    k = division(i, j);
15.    cout << k << endl;
16.  }catch (const char* e) {
17.    cerr << e << endl;
18.  }
19.  return 0;
20. }
```

Output:

```
Attempted to divide by zero!
```

C++ User-Defined Exceptions

The new exception can be defined by overriding and inheriting **exception** class functionality.

C++ user-defined exception example

Let's see the simple example of user-defined exception in which **std::exception** class is used to define the exception.

```
1. #include <iostream>
2. #include <exception>
3. using namespace std;
4. class MyException : public exception{
5.   public:
6.     const char * what() const throw()
7.     {
8.       return "Attempted to divide by zero!\n";
9.     }
10. };
11. int main()
```

```
{
    try
    {
        int x, y;
        cout << "Enter the two numbers : \n";
        cin >> x >> y;
        if (y == 0)
        {
            MyException z;
            throw z;
        }
        else
        {
            cout << "x / y = " << x/y << endl;
        }
    }
    catch(exception& e)
    {
        cout << e.what();
    }
}
```

Output:

```
Enter the two numbers :
10
2
x / y = 5
```

Output:

```
Enter the two numbers :
10
0
Attempted to divide by zero!
```

C++ try and catch

Exception handling in C++ consist of three keywords: try, throw and catch:

The try statement allows you to define a block of code to be tested for errors while it is being executed.

The throw keyword throws an exception when a problem is detected, which lets us create a custom error.

The catch statement allows you to define a block of code to be executed, if an error occurs in the try block.

TEMPLATES

Templates are powerful features of C++ which allows you to write generic programs. In simple terms, you can create a single function or a class to work with different data types using templates.

Templates are often used in larger codebase for the purpose of code reusability and flexibility of the programs.

The concept of templates can be used in two different ways:

- Function Templates
- Class Templates

Function Templates

A function template works in a similar to a normal function, with one key difference.

A single function template can work with different data types at once but, a single normal function can only work with one set of data types.

Example 1: Function Template to find the largest number

Program to display largest among two numbers using function templates.

```
1.  // If two characters are passed to function template, character with larger ASCII value is
    displayed.
2.
3.  #include <iostream>
4.  using namespace std;
5.
6.  // template function
7.  template <class T>
8.  T Large(T n1, T n2)
9.  {
10.         return (n1 > n2) ? n1 : n2;
11. }
12.
13. int main()
14. {
15.         int i1, i2;
16.         float f1, f2;
17.         char c1, c2;
18.
19.         cout << "Enter two integers:\n";
20.         cin >> i1 >> i2;
21.         cout << Large(i1, i2) <<" is larger." << endl;
22.
23.         cout << "\nEnter two floating-point numbers:\n";
24.         cin >> f1 >> f2;
25.         cout << Large(f1, f2) <<" is larger." << endl;
```

```
26.
27.         cout << "\nEnter two characters:\n";
28.         cin >> c1 >> c2;
29.         cout << Large(c1, c2) << " has larger ASCII value.";
30.
31.         return 0;
32. }
```

Output

```
Enter two integers:
5
10
10 is larger.

Enter two floating-point numbers:
12.4
10.2
12.4 is larger.

Enter two characters:
z
Z
z has larger ASCII value.
```

Class Templates

Like function templates, you can also create class templates for generic class operations.

Sometimes, you need a class implementation that is same for all classes, only the data types used are different.

Example 3: Simple calculator using Class template

Program to add, subtract, multiply and divide two numbers using class template

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  template <class T>
5.  class Calculator
6.  {
7.  private:
8.          T num1, num2;
9.
10. public:
11.         Calculator(T n1, T n2)
12.         {
```

```
13.                 num1 = n1;
14.                 num2 = n2;
15.         }
16.
17.         void displayResult()
18.         {
19.                 cout << "Numbers are: " << num1 << " and " << num2 << "." << endl;
20.                 cout << "Addition is: " << add() << endl;
21.                 cout << "Subtraction is: " << subtract() << endl;
22.                 cout << "Product is: " << multiply() << endl;
23.                 cout << "Division is: " << divide() << endl;
24.         }
25.
26.         T add() { return num1 + num2; }
27.
28.         T subtract() { return num1 - num2; }
29.
30.         T multiply() { return num1 * num2; }
31.
32.         T divide() { return num1 / num2; }
33. };
34.
35. int main()
36. {
37.         Calculator<int> intCalc(2, 1);
38.         Calculator<float> floatCalc(2.4, 1.2);
39.
40.         cout << "Int results:" << endl;
41.         intCalc.displayResult();
42.
43.         cout << endl << "Float results:" << endl;
44.         floatCalc.displayResult();
45.
46.         return 0;
47. }
```

Output

```
Int results:
Numbers are: 2 and 1.
Addition is: 3
Subtraction is: 1
Product is: 2
Division is: 2

Float results:
Numbers are: 2.4 and 1.2.
```

```
Addition is: 3.6
Subtraction is: 1.2
Product is: 2.88
Division is: 2
```

UNIT 5

Standard Template Library – Manipulating Strings – Object Oriented System Development

The C++ Standard Template Library (STL)

The Standard Template Library (STL) is a set of C++ template classes to provide common programming data structures and functions such as lists, stacks, arrays, etc.

It is a library of container classes, algorithms, and iterators.

It is a generalized library and so, its components are parameterized.

A working knowledge of template classes is a prerequisite for working with STL.

STL has four components

- Algorithms
- Containers
- Functions
- Iterators

Algorithms

The header algorithm defines a collection of functions especially designed to be used on ranges of elements.

They act on containers and provide means for various operations for the contents of the containers.

- Algorithm
  - Sorting
  - Searching
  - Important STL Algorithms
  - Useful Array algorithms
  - Partition Operations
- Numeric
  - valarray class

Containers

Containers or container classes store objects and data.

There are in total seven standard “first-class” container classes and three container adaptor classes and only seven header files that provide access to these containers or container adaptors.

- Sequence Containers: implement data structures which can be accessed in a sequential manner.
  - vector
  - list
  - deque
  - arrays
  - forward_list(Introduced in C++11)
- Container Adaptors : provide a different interface for sequential containers.
  - queue
  - priority_queue
  - stack
- Associative Containers : implement sorted data structures that can be quickly searched (O(log n) complexity).
  - set
  - multiset
  - map
  - multimap
- Unordered Associative Containers : implement unordered data structures that can be quickly searched
  - unordered_set (Introduced in C++11)
  - unordered_multiset (Introduced in C++11)
  - unordered_map (Introduced in C++11)
  - unordered_multimap (Introduced in C++11)

Functions

The STL includes classes that overload the function call operator. Instances of such classes are called function objects or functors.

Functors allow the working of the associated function to be customized with the help of parameters to be passed.

- Functors

Iterators

As the name suggests, iterators are used for working upon a sequence of values. They are the major feature that allow generality in STL.

- Iterators

Utility Library

Defined under <utility header>

- pair

C style string

The C style string belongs to C language and continues to support in C++ also strings in C are the one-dimensional array of characters which gets terminated by \0 (null character).

This is how the strings in C are declared:

```
char ch[6] = {'H', 'e', 'l', 'l', 'o', '
char ch[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
'};
```

Actually, you do not place the null character at the end of a string constant.

The C++ compiler automatically places the \0 at the end of the string when it initializes the array.

Manipulate Null-terminated strings

C++ supports a wide range of functions that manipulate null-terminated strings.

These are:

- strcpy(str1, str2): Copies string str2 into string str1.
- strcat(str1, str2): Concatenates string str2 onto the end of string str1.
- strlen(str1): Returns the length of string str1.
- strcmp(str1, str2): Returns 0 if str1 and str2 are the same; less than 0 if str1<str2; greater than 0 if str1>str2.
- strchr(str1, ch): Returns a pointer to the first occurrence of character ch in string str1.
- strstr(str1, str2): Returns a pointer to the first occurrence of string str2 in string str1.

Important functions supported by String Class

- append(): This function appends a part of a string to another string
- assign():This function assigns a partial string
- at(): This function obtains the character stored at a specified location
- begin(): This function returns a reference to the start of the string
- capacity(): This function gives the total element that can be stored
- compare(): This function compares a string against the invoking string

- empty(): This function returns true if the string is empty
- end(): This function returns a reference to the end of the string
- erase(): This function removes character as specified
- find(): This function searches for the occurrence of a specified substring
- length(): It gives the size of a string or the number of elements of a string
- swap(): This function swaps the given string with the invoking one

Object oriented system

We know that the Object-Oriented Modelling (OOM) technique visualizes things in an application by using models organized around objects.

Any software development approach goes through the following stages −

- Analysis,
- Design, and
- Implementation.

In object-oriented software engineering, the software developer identifies and organizes the application in terms of object-oriented concepts, prior to their final representation in any specific programming language or software tools.

Phases in Object-Oriented Software Development

The major phases of software development using object–oriented methodology are object-oriented analysis, object-oriented design, and object-oriented implementation.

Object–Oriented Analysis

In this stage, the problem is formulated, user requirements are identified, and then a model is built based upon real–world objects.

The analysis produces models on how the desired system should function and how it must be developed.

The models do not include any implementation details so that it can be understood and examined by any non–technical application expert.

Object–Oriented Design

Object-oriented design includes two main stages, namely, system design and object design.

System Design

In this stage, the complete architecture of the desired system is designed.

The system is conceived as a set of interacting subsystems that in turn is composed of a hierarchy of interacting objects, grouped into classes.

System design is done according to both the system analysis model and the proposed system architecture.

Here, the emphasis is on the objects comprising the system rather than the processes in the system.

Object Design

In this phase, a design model is developed based on both the models developed in the system analysis phase and the architecture designed in the system design phase.

All the classes required are identified. The designer decides whether −

- new classes are to be created from scratch,
- any existing classes can be used in their original form, or
- new classes should be inherited from the existing classes.

The associations between the identified classes are established and the hierarchies of classes are identified.

Besides, the developer designs the internal details of the classes and their associations, i.e., the data structure for each attribute and the algorithms for the operations.

Object–Oriented Implementation and Testing

In this stage, the design model developed in the object design is translated into code in an appropriate programming language or software tool.

The databases are created and the specific hardware requirements are ascertained. Once the code is in shape, it is tested using specialized techniques to identify and remove the errors in the code.