RAJIV GANDHI PROUDYOGIKI VISHWAVIDYALAYA, BHOPAL
New Scheme Based On AICTE Flexible Curricula
Information Technology, III-Semester

IT303 Object Oriented Programming & Methodology

Course Objectives

1. The objective of this course is to understand the advantage of object oriented programming over procedure oriented programming.
2. To help students to understand the key features of Object Oriented Programming and Methodology like objects, methods, instance, message passing, encapsulation, polymorphism, data hiding, abstract data and inheritance.
3. To develop understanding of pointers and memory management.
4. To be able to develop understanding of file input/output and templates

Unit I- Introduction: Object oriented programming, Introduction, Application, characteristics, difference between object oriented and procedure programming, Comparison of C and C++, Cout, Cin, Data Type, Type Conversion, Control Statement, Loops, Arrays and string arrays fundamentals, Function, Returning values from functions, Reference arguments, Overloaded function, Inline function, Default arguments, Returning by reference.

Unit II- Object and Classes: Implementation of class and object in C++, access modifiers, object as data type, constructor, destructor, Object as function arguments, default copy constructor, parameterized constructor, returning object from function, Structures and classes, Classes objects and memory, static class data, Arrays of object, Arrays as class Member Data, The standard C++ String class, Run time and Compile time polymorphism.

Unit III- Operator overloading and Inheritance: Overloading unary operators, Overloading binary operators, data conversion, pitfalls of operators overloading, Concept of inheritance, Derived class and base class, access modifiers, types of inheritance, Derived class constructors, member function, public and private inheritance.

Unit IV- Pointer and Virtual Function: Addresses and pointers, the address-of operator & pointer and arrays, Pointer and Function pointer, Memory management: New and Delete, pointers to objects, debugging pointers, Virtual Function, friend function, Static function, friend class, Assignment and copy initialization, this pointer, dynamic type information.

Unit V-Streams and Files: Streams classes, Stream Errors, Disk File I/O with streams, file pointers, error handling in file I/O with member function, overloading the extraction and insertion operators, memory as a stream object, command line arguments, printer output, Function templates, Class templates Exceptions, Containers, exception handling.

Course Outcomes

On the completion of this course students will be able to:
1. Recognize attributes and methods for given objects.
2. Define data types and also deal with operations applied for data structures.
3. Implement algorithms and complex problems.

Reference Books:

1. E. Balaguruswami, ”Object Oriented Programming in C++”, TMH.
2. Robert Lafore, “Object Oriented Programming in C++”, Pearson.
3. M.T. Somashekare, D.S. Guru, “ Object-Oriented Programming with C++”, PHI.
4. Herbert Shildt, “The Complete Reference C++”, Tata McGraw Hill publication.

List of Experiments:

1. Write a program to find out the largest number using function.
2. Write a program to find the area of circle, rectangle and triangle using function overloading.
3. Write a program to implement complex numbers using operator overloading and type conversion.
4. Write a program using class and object to print bio-data of the students.
5. Write a program which defines a class with constructor and destructor which will count number of object created and destroyed.
6. Write a program to implement single and multiple inheritances taking student as the sample base class.
7. Write a program to add two private data members using friend function.
8. Write a program using dynamic memory allocation to perform 2x2 matrix addition and subtraction.
9. Write a program to create a stack using virtual function.
10. Write a program that store five student records in a file.
11. Write a program to get IP address of the system.
12. Write a program to shutdown the system on windows operating system.

Class Notes

Object oriented programming

UNIT I

Object Oriented programming is a programming that is associated with the concept like Inheritance, Polymorphism, Abstraction, Encapsulation etc. and mainly emphasis on Class, Objects .

Object: Objects are basic run-time entities and they are instances of a class these are defined user defined data types.

Class: Class is a blueprint of data and functions or methods. Class does not take any space.

Data abstraction: It provides only essential information to the outside world and hiding their background details, i.e., to represent the needed information in program without presenting the details.
For example, a database system hides certain details of how data is stored and created and maintained. Similar way, C++ classes provides different methods to the outside world without giving internal detail about those methods and data.

Encapsulation: Encapsulation is placing the data and the functions that work on that data in the same place. While working with procedural languages, it is not always clear which functions work on which variables but object-oriented programming provides you framework to place the data and the relevant functions together in the same object.

Inheritance: Inheritance is the process by which objects of one class acquire the properties of objects of another class. It supports the concept of hierarchical classification. Inheritance provides re usability. Inheritance is the process of forming a new class from an existing class that is from the existing class called as base class, new class is formed called as derived class.
This is a very important concept of object-oriented programming since this feature helps to reduce the code size.

Polymorphism: Polymorphism means ability to take more than one form. An operation may exhibit different behaviors in different instances. The behavior depends upon the types of data used in the operation.

Dynamic Binding: In dynamic binding, the code to be executed in response to function call is decided at runtime. C++ has virtual functions to support this.

Message Passing: Objects communicate with one another by sending and receiving information to each other. A message for an object is a request for execution of a procedure and therefore will invoke a function in the receiving object that generates the desired results. Message passing involves specifying the name of the object, the name of the function and the information to be sent.

Difference between Procedure Oriented Programming (POP) & Object Oriented Programming (OOP)

Object Oriented Programming	Procedure Oriented Programming	Points
In OOP, program is divided into parts called objects.	In POP, program is divided into small parts called functions.	Divided Into
In OOP, Importance is given to the data rather than procedures or functions because it works as a real world.	In POP, Importance is not given to data but to functions as well as sequence of actions to be done.	Importance
OOP follows Bottom Up approach.	POP follows Top Down approach.	Approach

OOP has access specifiers named Public, Private, Protected, etc.	POP does not have any access specifier.	Access Specifiers
In OOP, objects can move and communicate with each other through member functions.	In POP, Data can move freely from function to function in the system.	Data Moving
OOP provides an easy way to add new data and function.	To add new data and function in POP is not so easy.	Expansion
In OOP, data cannot move easily from function to function, it can be kept public or private so we can control the access of data.	In POP, Most function uses Global data for sharing that can be accessed freely from function to function in the system.	Data Access
OOP provides Data Hiding so provides more security.	POP does not have any proper way for hiding data so it is less secure.	Data Hiding
In OOP, overloading is possible in the form of Function Overloading and Operator Overloading.	In POP, Overloading is not possible.	Overloading
Example of OOP are : C++, JAVA, VB.NET, C#.NET.	Example of POP are : C, VB, FORTRAN, Pascal.	Examples

Local Environment Setup
If you are still willing to set up your environment for C++, you need following two softwares available on your computer.
Text Editor:
This will be used to type your program. Examples of few editors include Windows Notepad, OS Edit command, Brief, Epsilon, EMACS, and vim or vi.
Name and version of text editor can vary on different operating systems. For example, Notepad will be used on Windows and vim or vi can be used on windows as well as Linux, or UNIX.
The files you create with your editor are called source files and for C++ they typically are named with the extension .cpp, .cp, or .c.
Before starting your programming, make sure you have one text editor in place and you have enough experience to type your C++ program.
C++ Compiler:
This is actual C++ compiler, which will be used to compile your source code into final executable program.
Most C++ compilers don't care what extension you give your source code, but if you don't specify otherwise, many will use .cpp by default
Most frequently used and free available compiler is GNU C/C++ compiler; otherwise you can have compilers either from HP or Solaris if you have respective Operating Systems.
Installing GNU C/C++ Compiler:

Difference between C & C++

Sl. No	Basis Of Distinction	C	C++
1	**Nature Of Language**	C is a structural or procedural type of programming language.	C++ is an object-oriented programming language and supports Polymorphism, Abstract Data Types, Encapsulation, among others. Even though C++ derives basic syntax from C, it cannot be classified as a structural or a procedural language.
2	**Point Of Emphasis**	C lays emphasis on the steps or procedures that are followed to solve a problem.	C++ emphasizes the objects and not the steps or procedures. It has higher abstraction level.
3	**Compatibility With Overloading**	C does not support function overloading.	C++ supports function overloading, implying that one can have name of functions with varying parameters.
4	**Data Types**	C does not provide String or Boolean data types. It supports primitive & built-in data types.	C++ provides Boolean or String data types. It supports both user-defined and built-in data types.
5	**Compatibility With Exception Handling**	C does not support Exception Handling directly. It can be done through some other functions.	C++ supports Exception: Handling can be done through try & catch block.
6	**Compatibility With Functions**	C does not support functions with default arrangements	C++ supports functions with default arrangements.
7	**Compatibility With Generic Programming**	C is not compatible	C++ is compatible with generic programming

8	**Pointers And References**	C supports only Pointers	C++ supports both pointers and references.
9	**Inline Function**	C does not have inline function.	C++ has inline function.
10	**Data Security**	In C programming language, the data is unsecured.	Data is hidden in C++ and is not accessible to external functions. Hence, is more secure
11	**Approach**	C follows the top-down approach.	C++ follows the bottom-up approach.
12	**Functions For Standard Input And Output**	scanf and printf	cin and cout
13	**Time Of Defining Variables**	In C, variable has to be defined at the beginning, in the function.	Variable can be defined anywhere in the function.
14	**Namespace**	Absent	Present
15	**Division Of Programs**	The programs in C language are divided into modules and functions.	The programs are divided into classes and functions in the C++ programming language.
16	**File Extension**	.C	.CPP
17	**Function And Operator Overloading**	Absent	Present
18	**Mapping**	Mapping between function and data is complicated in C.	Mapping between function and data can be done easily using 'Objects'.

19	**Calling Of Functions**	main() function can be called through other functions.	main() function cannot be called through other functions.
20	**Inheritance**	Possible	Not possible
21	**Influences**	C++, C#, Objective-C, PHP, Perl, BitC, Concurrent C, Java, JavaScript	C#, PHP, Java, D, Aikido, Ada 95

C++ Basic Input/Output
C++ I/O occurs in streams, which are sequences of bytes. If bytes flow from a device like a keyboard, a disk drive, or a network connection etc. to main memory, this is called **input operation** and if bytes flow from main memory to a device like a display screen, a printer, a disk drive, or a network connection, etc., this is called **output operation**.

The Standard Output Stream (cout)
The predefined object cout is an instance of ostream class. The cout object is said to be "connected to" the standard output device, which usually is the display screen. The cout is used in conjunction with the stream insertion operator, which is written as << which are two less than signs as shown in the following example.

```
#include <iostream>
using namespace std;
int main() {
charstr[] = "Hello C++";
cout<< "Result of str is : " <<str<<endl;
}
```

When the above code is compiled and executed, it produces the following result −

```
Value of stris : Hello C++
```

The C++ compiler also determines the data type of variable to be output and selects the appropriate stream insertion operator to display the value. The << operator is overloaded to output data items of built-in types integer, float, double, strings and pointer values.
The insertion operator << may be used more than once in a single statement as shown above and endl is used to add a new-line at the end of the line.

The Standard Input Stream (cin)
The predefined object cin is an instance of istream class. The cin object is said to be attached to the standard input device, which usually is the keyboard. The cin is used in conjunction with the stream extraction operator, which is written as >> which are two greater than signs as shown in the following example.

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

```
char name[50];
cout<< "Please enter your name: ";
cin>> name;
cout<< "Your name is: " << name <<endl;
}
```

When the above code is compiled and executed, it will prompt you to enter a name. You enter a value and then hit enter to see the following result −

```
Please enter your name: cplusplus
Your name is: cplusplus
```

The C++ compiler also determines the data type of the entered value and selects the appropriate stream extraction operator to extract the value and store it in the given variables.
The stream extraction operator >> may be used more than once in a single statement.

Data Type
Primitive Built-in Types
C++ offer the programmer a rich assortment of built-in as well as user defined data types. Following table lists down seven basic C++ data types:

Keyword	Type
bool	Boolean
char	Character
int	Integer
float	Floating point
double	Double floating point
void	Valueless
wchar_t	Wide character

Several of the basic types can be modified using one or more of these type modifiers:
signed
unsigned
short
long

The following table shows the variable type, how much memory it takes to store the value in memory, and what is maximum and minimum value which can be stored in such type of variables.

Typical Range	Typical Bit Width	Type
-128 to 127 or 0 to 255	1byte	char
0 to 255	1byte	unsigned char

-128 to 127	1byte	signed char
-2147483648 to 2147483647	4bytes	int
0 to 4294967295	4bytes	unsigned int
-2147483648 to 2147483647	4bytes	signed int
-32768 to 32767	2bytes	short int
0 to 65,535	2bytes	unsigned short int
-32768 to 32767	2bytes	signed short int
-2,147,483,648 to 2,147,483,647	8bytes	long int
-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	8bytes	signed long int
0 to 18,446,744,073,709,551,615	8bytes	unsigned long int
+/- 3.4e +/- 38 (~7 digits)	4bytes	float
+/- 1.7e +/- 308 (~15 digits)	8bytes	double

The sizes of variables might be different from those shown in the above table, depending on the compiler and the computer you are using.
Following is the example, which will produce correct size of various data types on your computer.

```
#include<iostream>
usingnamespacestd;
int main(){
cout<<"Size of char : "<<sizeof(char)<<endl;
cout<<"Size of int : "<<sizeof(int)<<endl;
cout<<"Size of short int : "<<sizeof(shortint)<<endl;
cout<<"Size of long int : "<<sizeof(longint)<<endl;
cout<<"Size of float : "<<sizeof(float)<<endl;
cout<<"Size of double : "<<sizeof(double)<<endl;
cout<<"Size of wchar_t : "<<sizeof(wchar_t)<<endl;
return0;
}
```

This example uses endl, which inserts a new-line character after every line and << operator is being used to pass multiple values out to the screen. We are also using sizeof() operator to get size of various data types.
When the above code is compiled and executed, it produces the following result which can vary from machine to machine –

```
Size of char : 1
Size of int : 4
Size of short int : 2
Size of long int : 8
Size of float : 4
Size of double : 8
Size of wchar_t : 4
```

There are following basic types of variable in C++ .

Type	Description
bool	Stores either value true or false.
char	Typically a single octet(one byte). This is an integer type.
int	The most natural size of integer for the machine.
float	A single-precision floating point value.
double	A double-precision floating point value.
void	Represents the absence of type.
wchar_t	A wide character type.

Variable Declaration in C++

A variable declaration has its meaning at the time of compilation only, compiler needs actual variable definition at the time of linking of the program. A variable declaration provides assurance to the compiler that there is one variable existing with the given type and name so that compiler proceed for further compilation without needing complete detail about the variable.

A variable declaration is useful when you are using multiple files and you define your variable in one of the files which will be available at the time of linking of the program. You will use extern keyword to declare a variable at any place. Though you can declare a variable multiple times in your C++ program, but it can be defined only once in a file, a function or a block of code.

Example

Try the following example where a variable has been declared at the top, but it has been defined inside the main function:

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

// Variable declaration:
externint a, b;
externint c;
extern float f;

int main () {
  // Variable definition:
int a, b;
int c;
float f;

  // actual initialization
  a = 20;
  b = 30;
  c = a + b;
cout<< c <<endl ;
   f = 70.0/3.0;
cout<< f <<endl ;
return 0;
}
```
When the above code is compiled and executed, it produces the following result −
```
50
23.3333
```

Local Variables

Variables that are declared inside a function or block are local variables. They can be used only by statements that are inside that function or block of code. Local variables are not known to functions outside their own. Following is the example using local variables:

```
#include <iostream>
using namespace std;
int main () {
   // Local variable declaration:
int a, b;
int c;
   // actual initialization
   a = 50;
   b =320;
   c = a + b;
cout<< c;
return 0;
}
```
Output
```
80
```

Global Variables

Global variables are defined outside of all the functions, usually on top of the program. The global variables will hold their value throughout the life-time of your program.

A global variable can be accessed by any function. That is, a global variable is available for use throughout your entire program after its declaration. Following is the example using global and local variables:

```
#include <iostream>
using namespace std;
// Global variable declaration:
int global;
int main () {
   // Local variable declaration:
int a, b;
   // actual initialization
   a = 10;
   b = 20;
global= a + b;
cout<<global;
return 0;
}
```
Output
```
80
```
A program can have same name for local and global variables but value of local variable inside a function will take preference. For example:

```
#include <iostream>
using namespace std;
// Global variable declaration:
intglobal = 20;
int main () {
  // Local variable declaration:
intglobal = 10;
cout<<global;
return 0;
}
```
When the above code is compiled and executed, it produces the following result:
```
10
```

CONTROL STATEMENT

The Control Statements are used for controlling the Execution of the program there are the three control statements those are break, goto, continue.

A control statement is a statement that determines whether other statements will be executed.

- An if statement decides whether to execute another statement, or decides which of two statements to execute.
- A loop decides how many times to execute another statement. There are three kinds of loops:
- while loops test whether a condition is true before executing the controlled statement.
- do-while loops test whether a condition is true after executing the controlled statement.
- for loops are (typically) used to execute the controlled statement a given number of times.
- A switch statement decides which of several statements to execute

  There are two kinds of if statements--those with an else clause, and those without an else clause.

  Without an else clause

An if statement tests a condition. If the condition is true, the following statement (typically, a block) is executed. If the condition is not true, the if statement does nothing. The syntax is:

```
if (condition) {
statements
}
```

For example,

```
if (x < 0) {
   x = 0;
}
```

The above if statement resets x to zero if it has become negative.

The braces indicate a block of statements. If there is only one statement, the braces may be omitted; however, it is good style to always include the braces (reasons are given below).

With an else clause

An if-else statement tests a condition, and chooses one of two statements to execute. If the condition is true, the statement following the condition is executed. If the condition is not true, the statement following the else is executed. Both statements are typically blocks. The syntax is:

```
if (condition) {
   some statements
}
else {
   some other statements
}
```

For example,

1. If you leave out the braces around the "true part", the semicolon makes it look like the else is a separate statement. It is not. There is no such thing as an "else statement."

```
if (x == 0)
System.out.println("x is zero");
else {
System.out.println("x is not zero");
}
```

2. If you leave out the braces, it is easy to make the following error:

```
if (x < 0)
System.out.println("x reset to zero");
   x = 0;
System.out.println("x is now " + x); // always prints 0
```

The indentation suggests that two statements are under control of the if statement, but in fact only the first one is.

For both of the above reasons, it is good style to always include the braces (for both the if part and the else part), even when they enclose only a single statement.

3. If you have an else part, it is usually better to use a "positive" condition than a negated one:

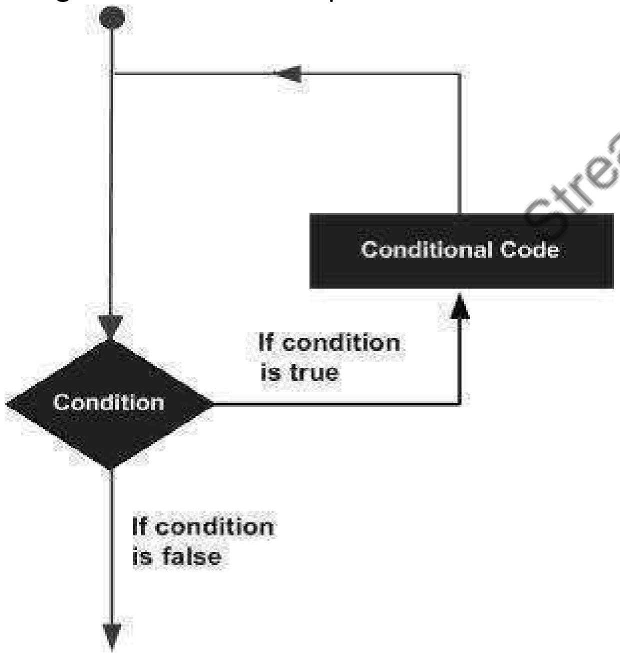
	Better style	Poor style

	if (danger) {
	System.out.println("Run away!");
	}
	else {
	System.out.println("Relax.");
		if (!danger) {
		System.out.println("Relax.");
		}
		else {
		System.out.println("Run away!");

	}	}

	if (x == 100) {
	System.out.println("Perfect score!");
	}
	else {
	System.out.println("You could do better.");
	}
		if (x != 100) {
		System.out.println("You could do better.");
		}
		else {
		System.out.println("Perfect score!");
		}

4. This is so that when you read your program, you don't have to think "Let's see--it's *not* the case that danger is false, so that means there *is* danger, so I should run away." Or, "If I didn't *not* get 100, I got a perfect score."
   There may be a situation, when you need to execute a block of code several number of times. In general statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.
   Programming languages provide various control structures that allow for more complicated execution paths.
   A loop statement allows us to execute a statement or group of statements multiple times and following is the general from of a loop statement in most of the programming languages:



C++ programming language provides the following types of loop to handle looping requirements. Click the following links to check their detail.

Description	Loop Type
Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.	while loop

Execute a sequence of statements multiple times and abbreviates the code that manages the loop variable.	for loop
Like a while statement, except that it tests the condition at the end of the loop body	do...while loop
You can use one or more loop inside any another while, for or do..while loop.	nested loops

Loops
Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.
C++ supports the following control statements. Click the following links to check their detail.

Functions:-
A function is a group of statements that together perform a task. Every C++ program has at least one function, which is main (), and all the most trivial programs can define additional functions.

You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division usually is so each function performs a specific task.
A function declaration tells the compiler about a function's name, return type, and parameters. A function definition provides the actual body of the function.
The C++ standard library provides numerous built-in functions that your program can call. For example, function strcat() to concatenate two strings, function memcpy() to copy one memory location to another location and many more functions.
A function is knows as with various names like a method or a sub-routine or a procedure etc.

Defining a Function
The general form of a C++ function definition is as follows:
return_typefunction_name(parameter list) {
body of the function
}
A C++ function definition consists of a function header and a function body. Here are all the parts of a function:
Return Type: A function may return a value. The return_type is the datatype of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword void.

Function Name: This is the actual name of the function. The function name and the parameter list together constitute the function signature.
Parameters: A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.

Function Body: The function body contains a collection of statements that define what the function does.

Example

Following is the source code for a function called max(). This function takes two parameters num1 and num2 and returns the maximum between the two:

```
// function returning the max between two numbers
int max(int num1, int num2)  {
   // local variable declaration
int result;

if (num1 > num2)
result = num1;
else
result = num2;

return result;
}
```

Function Declarations

A function declaration tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately.

A function declaration has the following parts:

```
return_typefunction_name( parameter list );
```

For the above defined function max(), following is the function declaration:

```
int max(int num1, int num2);
```

Parameter names are not important in function declaration only their type is required, so following is also valid declaration:

```
int max(int, int);
```

Function declaration is required when you define a function in one source file and you call that function in another file. In such case, you should declare the function at the top of the file calling the function.

Calling a Function

While creating a C++ function, you give a definition of what the function has to do. To use a function, you will have to call or invoke that function.

When a program calls a function, program control is transferred to the called function. A called function performs defined task and when its return statement is executed or when its function-ending closing brace is reached, it returns program control back to the main program.

To call a function, you simply need to pass the required parameters along with function name, and if function returns a value, then you can store returned value. For example:

```
#include <iostream>
using namespace std;
// function declaration
int max(int num1, int num2);
int main () {
   // local variable declaration:
int a = 100;
int b = 200;
int ret;
   // calling a function to get max value.
ret = max(a, b);
cout<< "Max value is : " << ret <<endl;
return 0;
}
// function returning the max between two numbers
int max(int num1, int num2)  {
   // local variable declaration
int result;
if (num1 > num2)
result = num1;
else
result = num2;
return result;
}
```

I kept max() function along with main() function and compiled the source code. While running final executable, it would produce the following result:

```
Max value is : 200
```

Function Arguments

If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the formal parameters of the function.

The formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.

While calling a function, there are two ways that arguments can be passed to a function:

Call Type Description

Call by value

This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.

Call by pointer

This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

Call by reference

This method copies the reference of an argument into the formal parameter. Inside the function, the reference is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

By default, C++ uses call by value to pass arguments. In general, this means that code within a function cannot alter the arguments used to call the function and above mentioned example while calling max() function used the same method.

Default Values for Parameters

When you define a function, you can specify a default value for each of the last parameters. This value will be used if the corresponding argument is left blank when calling to the function.

This is done by using the assignment operator and assigning values for the arguments in the function definition. If a value for that parameter is not passed when the function is called, the default given value is used, but if a value is specified, this default value is ignored and the passed value is used instead. Consider the following example:

```
#include <iostream>
using namespace std;
intadd(int a, int b=20)
 {
int result;
result = a + b;
return (result);
}

int main () {
   // local variable declaration:
int a = 100;
int b = 200;
int result;

   // calling a function to add the values.
result = add(a, b);
cout<< "Total value is :" << result <<endl;

   // calling a function again as follows.
result = add(a);
cout<< "Total value is :" << result <<endl;
return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Total value is :300
Total value is :120
```

Arrays:-

C++ provides a data structure, the array, which stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables. A specific element in an array is accessed by an index.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.

Declaring Arrays

To declare an array in C++, the programmer specifies the type of the elements and the number of elements required by an array as follows:

typearrayName [arraySize];
This is called a single-dimension array. The arraySize must be an integer constant greater than zero and type can be any valid C++ data type. For example, to declare a 10-element array called balance of type double, use this statement:

double balance[10];

Initializing Arrays
You can initialize C++ array elements either one by one or using a single statement as follows:
double balance[5] = {1000.0, 2.0, 3.4, 17.0, 50.0};
The number of values between braces { } cannot be larger than the number of elements that we declare for the array between square brackets [ ]. Following is an example to assign a single element of the array:

If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write:

double balance[] = {1000.0, 2.0, 3.4, 17.0, 50.0};
You will create exactly the same array as you did in the previous example.

balance[4] = 50.0;
The above statement assigns element number 5th in the array a value of 50.0. Array with 4th index will be 5th, i.e., last element because all arrays have 0 as the index of their first element which is also called base index. Following is the pictorial representation of the same array we discussed above:

Array Presentation
Accessing Array Elements
An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array. For example:
double salary = balance[9];

The above statement will take 10th element from the array and assign the value to salary variable. Following is an example, which will use all the above-mentioned three concepts viz. declaration, assignment and accessing arrays:

```
#include <iostream>
using namespace std;
#include <iomanip>
usingstd::setw;
int main () {
int n[ 10 ]; // n is an array of 10 integers

   // initialize elements of array n to 0
for ( int i = 0; i < 10; i++ ) {
n[ i ] = i + 100; // set element at location i to i + 100
   }

cout<< "Element" <<setw( 13 ) << "Value" <<endl;
   // output each array element's value
for ( int j = 0; j < 10; j++ ) {
```

```
cout<<setw( 7 )<< j <<setw( 13 ) << n[ j ] <<endl;
   }
Return 0;
}
```

This program makes use of setw() function to format the output. When the above code is compiled and executed, it produces the following result:

```
Element       Value
     0         100
     1         101
     2         102
     3         103
     4         104
     5         105
     6         106
     7         107
     8         108
     9         109
```

C++ Arrays
Arrays are important to C++ and should need lots of more detail. There are following few important concepts, which should be clear to a C++ programmer:

Multi-dimensional arrays
C++ supports multidimensional arrays. The simplest form of the multidimensional array is the two-dimensional array.

Pointer to an array
You can generate a pointer to the first element of an array by simply specifying the array name, without any index.

Passing arrays to functions
You can pass to the function a pointer to an array by specifying the array's name without an index.

Return array from functions
C++ allows a function to return an array.

Function Overloading
The same name of function but parameters are different is called function overloading .

An overloaded declaration is a declaration that had been declared with the same name as a previously declared declaration in the same scope, except that both declarations have different arguments and obviously different definition (implementation).

When you call an overloaded function or operator, the compiler determines the most appropriate definition to use by comparing the argument types you used to call the function or operator with the parameter types specified in the definitions. The process of selecting the most appropriate overloaded function or operator is called overload resolution.
You can have multiple definitions for the same function name in the same scope. The definition of the function must differ from each other by the types and/or the number of arguments in the argument list.

You cannot overload function declarations that differ only by return type.

Following is the example where same function show() is being used to print different data types:

```
#include <iostream>
using namespace std;
classData
{
public:
voidshow(int i) {
cout<< "Printing int: " << i <<endl;
      }

voidshow(double  f) {
cout<< "Printing float: " << f <<endl;
      }

voidshow(char* c) {
cout<< "Printing character: " << c <<endl;
      }
};
int main(void) {
Datapd;

   // Call print to print integer
pd.show(5);

   // Call print to print float
pd.show(500.263);

   // Call print to print character
pd.show("Hello C++");

return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Printing int: 5
Printing float: 500.263
Printing character: Hello C++
```

Operators overloading in C++

You can redefine or overload most of the built-in operators available in C++. Thus a programmer can use operators with user-defined types as well.

Overloaded operators are functions with special names the keyword operator followed by the symbol for the operator being defined. Like any other function, an overloaded operator has a return type and a parameter list.

Box operator+(const Box&);
declares the addition operator that can be used to add two Box objects and returns final Box object. Most overloaded operators may be defined as ordinary non-member functions or as class member functions. In case we define above function as non-member function of a class then we would have to pass two arguments for each operand as follows:

Box operator+(const Box&, const Box&);
Following is the example to show the concept of operator over loading using a member function. Here an object is passed as an argument whose properties will be accessed using this object, the object which will call this operator can be accessed using this operator as explained below:

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

classContainer{
public:

doublegetVolume(void) {
return length * breadth * height;
    }

voidsetLength( double len ) {
length = len;
    }

voidsetBreadth( double bre ) {
breadth = bre;
    }

voidsetHeight( double hei ) {
height = hei;
    }

    // Overload + operator to add two Box objects.
Container operator+(constContainer& b) {
Container box;
box.length = this->length + b.length;
box.breadth = this->breadth + b.breadth;
box.height = this->height + b.height;
return box;
    }

private:
double length;      // Length of a box
double breadth;     // Breadth of a box
double height;      // Height of a box
};

// Main function for the program
```

```
int main( ) {
ContainerContainer1;              // Declare Container1
ContainerContainer2;              // Declare Container2
ContainerContainer3;              // Declare Container3
double volume = 0.0;     // Store the volume of a box here

   // Container 1 specification
Container1.setLength(6.0);
Container1.setBreadth(7.0);
Container1.setHeight(5.0);

   // Container 2 specification
Container2.setLength(12.0);
Container2.setBreadth(13.0);
Container2.setHeight(10.0);

   // volume of Container1
volume = Container1.getVolume();
cout<< "Volume of Container1 : " << volume <<endl;

   // volume of Container2
volume = Container2.getVolume();
cout<< "Volume of Container2 : " << volume <<endl;

   // Add two object as follows:
Container3 = Container1 + Container2;

   // volume of Container3
volume = Container3.getVolume();
cout<< "Volume of Container3 : " << volume <<endl;

return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Volume of Container1 : 210
Volume of Container2 : 1560
Volume of Container3 : 5400
```

Overloadable/Non-overloadableOperators
Following is the list of operators which can be overloaded:

+	-	*	/	%	^
&	|	~	!	,	=
<	>	<=	>=	++	--
<<	>>	==	!=	&&	||
+=	-=	/=	%=	^=	&=
|=	*=	<<=	>>=	[]	()

->	->*	new	new []	delete	delete []

Following is the list of operators, which cannot be overloaded:

::	.*	.	?:

Operator Overloading Examples

Here are various operator overloading examples to help you in understanding the concept.

S.N.	Operators and Example
1	Unary operators overloading
2	Binary operators overloading
3	Relational operators overloading
4	Input/output operators overloading
5	++ and -- operators overloading
6	Assignment operators overloading
7	Function call () operator overloading
8	Subscripting [] operator overloading
9	Class member access operator -> overloading

UNIT II

The main purpose of C++ programming is to add object orientation to the C programming language and classes are the central feature of C++ that supports object-oriented programming and are often called user-defined types.

A class is used to specify the form of an object and it combines data representation and methods for manipulating that data into one neat package. The data and functions within a class are called members of the class.

C++ Class Definitions

When you define a class, you define a blueprint for a data type. This doesn't actually define any data, but it does define what the class name means, that is, what an object of the class will consist of and what operations can be performed on such an object.

A class definition starts with the keyword class followed by the class name; and the class body, enclosed by a pair of curly braces. A class definition must be followed either by a semicolon or a list of declarations. For example, we defined the Box data type using the keyword class as follows:

```
class Box {
public:
double length; // Length of a box
double breadth;  // Breadth of a box
double height;  // Height of a box
};
```

The keyword public determines the access attributes of the members of the class that follow it. A public member can be accessed from outside the class anywhere within the scope of the class object. You can also specify the members of a class as private or protected which we will discuss in a sub-section.

Define C++ Objects

A class provides the blueprints for objects, so basically an object is created from a class. We declare objects of a class with exactly the same sort of declaration that we declare variables of basic types. Following statements declare two objects of class Box:

```
Box Box1;         // Declare Box1 of type Box
Box Box2;         // Declare Box2 of type Box
```

Both of the objects Box1 and Box2 will have their own copy of data members.

Access Modifiers in C++

Access modifiers are used to implement important feature of Object Oriented Programming known as Data Hiding. Consider a real life example: What happens when a driver applies brakes? The car stops. The driver only knows that to stop the car, he needs to apply the brakes. He is unaware of how actually the car stops. That is how the engine stops working or the internal implementation on the engine side. This is what data hiding is.

Access modifiers or Access Specifiers in a class are used to set the accessibility of the class members. That is, it sets some restrictions on the class members not to get directly accessed by the outside functions.

There are 3 types of access modifiers available in C++:

- Public
- Private
- Protected

Note: If we do not specify any access modifiers for the members inside the class then by default the access modifier for the members will be Private.

Public: All the class members declared under public will be available to everyone. The data members and member functions declared public can be accessed by other classes too. The public members of a class can be accessed from anywhere in the program using the direct member access operator (.) with the object of that class.

```
// C++ program to demonstrate public
// access modifier
#include<iostream>
using namespace std;
// class definition
class Circle
{
public:
double radius;
double  compute_area()
       {
return 3.14*radius*radius;
       }

};
 // main function
int main()
{
   Circle obj;
  // accessing public datamember outside class
obj.radius = 5.5;
cout<< "Radius is:" <<obj.radius<< "\n";
cout<< "Area is:" <<obj.compute_area();
return 0;
}
```

Output:

```
Radius is:5.5
Area is:94.985
```

In the above program the data member radius is public so we are allowed to access it outside the class.

Private: The class members declared as private can be accessed only by the functions inside the class. They are not allowed to be accessed directly by any object or function outside the class. Only the member functions or the friend functions are allowed to access the private data members of a class.

Example:

```
// C++ program to demonstrate private
// access modifier
#include<iostream>
using namespace std;
class Circle
{
   // private data member
private:
```

```
double radius;
      // public member function
public:
double  compute_area()
       {  // member function can access private
           // data member radius
return 3.14*radius*radius;
       }
    };
 // main function
int main()
{
   // creating object of the class
   Circle obj;
        // trying to access private data member
   // directly outside the class
obj.radius = 1.5;
cout<< "Area is:" <<obj.compute_area();
return 0;
}
```

The output of above program will be a compile time error because we are not allowed to access the private data members of a class directly outside the class.

Output:

```
 In function 'intmain()':
11:16: error: 'double Circle::radius' is private
double radius;
               ^
31:9: error: within this context
obj.radius = 1.5;
         ^
```

However we can access the private data members of a class indirectly using the public member functions of the class. Below program explains how to do this:

```
// C++ program to demonstrate private
// access modifier
#include<iostream>
using namespace std;
class Circle
{
   // private data member
private:
double radius;
 // public member function
public:
double  compute_area(double r)
       {  // member function can access private
           // data member radius
radius = r;
double area = 3.14*radius*radius;
cout<< "Radius is:" << radius <<endl;
cout<< "Area is: " << area;
        }
      };

// main function
int main()
{       // creating object of the class
    Circle obj;
         // trying to access private data member
    // directly outside the class
obj.compute_area(1.5);
return 0;
}
```

Output:

```
Radius is:1.5
Area is: 7.065
```

Protected: Protected access modifier is similar to that of private access modifiers, the difference is that the class member declared as Protected are inaccessible outside the class but they can be accessed by any subclass(derived class) of that class.

```
// C++ program to demonstrate
// protected access modifier
#include <bits/stdc++.h>
using namespace std;
// base class
class Parent
{
    // protected data members
protected:
intid_protected;
  };
// sub class or derived class
class Child : public Parent
{
public:
voidsetId(int id)
    {
         // Child class is able to access the inherited
        // protected data members of base class
id_protected = id;
            }
voiddisplayId()
    {
cout<< "id_protected is:" <<id_protected<<endl;
    }
};
```

```
// main function
int main() {
       Child obj1;
     // member function of derived class can
    // access the protected data members of base class
obj1.setId(81);
obj1.displayId();
return 0;
}
```

Output:
```
id_protected is:81
```

Accessing the Data Members
The public data members of objects of a class can be accessed using the direct member access operator (.). Let us try the following example to make the things clear:

```
#include <iostream>
using namespace std;
class Box {
public:
double length; // Length of a box
double breadth;  // Breadth of a box
double height;  // Height of a box
};

int main( ) {
   Box Box1;        // Declare Box1 of type Box
   Box Box2;        // Declare Box2 of type Box
double volume = 0.0;     // Store the volume of a box here
   // box 1 specification
   Box1.height = 5.0;
   Box1.length = 6.0;
   Box1.breadth = 7.0;
   // box 2 specification
   Box2.height = 10.0;
   Box2.length = 12.0;
   Box2.breadth = 13.0;
   // volume of box 1
volume = Box1.height * Box1.length * Box1.breadth;
cout<< "Volume of Box1 : " << volume <<endl;

   // volume of box 2
volume = Box2.height * Box2.length * Box2.breadth;
cout<< "Volume of Box2 : " << volume <<endl;

return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Volume of Box1 : 210
Volume of Box2 : 1560
```

It is important to note that private and protected members can not be accessed directly using direct member access operator (.). We will learn how private and protected members can be accessed.

Classes & Objects in Detail

So far, you have got very basic idea about C++ Classes and Objects. There are further interesting concepts related to C++ Classes and Objects which we will discuss in various sub-sections listed below:

Concept Description

Class member functions

A member function of a class is a function that has its definition or its prototype within the class definition like any other variable.

Class access modifiers

A class member can be defined as public, private or protected. By default members would be assumed as private.

Constructor & destructor

A class constructor is a special function in a class that is called when a new object of the class is created. A destructor is also a special function which is called when created object is deleted.

C++ copy constructor

The copy constructor is a constructor which creates an object by initializing it with an object of the same class, which has been created previously.

C++ friend functions

A friend function is permitted full access to private and protected members of a class.

C++ inline functions

With an inline function, the compiler tries to expand the code in the body of the function in place of a call to the function.

The this pointer in C++

Every object has a special pointer this which points to the object itself.

Pointer to C++ classes

A pointer to a class is done exactly the same way a pointer to a structure is. In fact a class is really just a structure with functions in it.

Static members of a class

Both data members and function members of a class can be declared as static.

Constructor & Destructor
Constructor
It is a member function having same name as it's class and which is used to initialize the objects of that class type with a legel initial value. Constructor is automatically called when object is created.

Types of Constructor
Default Constructor-: A constructor that accepts no parameters is known as default constructor. If no constructor is defined then the compiler supplies a default constructor.

```
Circle :: Circle()
{
radius = 0;
}
```
Parameterized Constructor -: A constructor that receives arguments/parameters, is called parameterized constructor.

```
Circle :: Circle(double r)
{
radius = r;
}
```
Copy Constructor-: A constructor that initializes an object using values of another object passed to it as parameter, is called copy constructor. It creates the copy of the passed object.
```
Circle :: Circle(Circle &t)
{
radius = t.radius;
}
```

There can be multiple constructors of the same class, provided they have different signatures.
Destructor
A destructor is a member function having sane name as that of its class preceded by ~(tilde) sign and which is used to destroy the objects that have been created by a constructor. It gets invoked when an object's scope is over.
```
~Circle() {}
```

Example : In the following program constructors, destructor and other member functions are defined inside class definitions. Since we are using multiple constructor in class so this example also illustrates the concept of constructor overloading
```
#include<iostream>
using namespace std;
class Circle //specify a class
{
private :
double radius; //class data members
public:
Circle() //default constructor
```

```
        {
radius = 0;
        }
Circle(double r) //parameterized constructor
        {
radius = r;
        }
Circle(Circle &t) //copy constructor
        {
radius = t.radius;
        }
voidsetRadius(double r) //function to set data
        {
radius = r;
        }
doublegetArea()
        {
return 3.14 * radius * radius;
        }
        ~Circle() //destructor
        {}
};
int main()
{
   Circle c1; //defalut constructor invoked
   Circle c2(2.5); //parmeterized constructor invoked
   Circle c3(c2); //copy constructor invoked
cout<< c1.getArea()<<endl;
cout<< c2.getArea()<<endl;
cout<< c3.getArea()<<endl;
return 0;
}
```

Another way of Member initialization in constructors
The constructor for this class could be defined, as usual, as:

```
Circle :: Circle(double r)
{
radius = r;
}
```

It could also be defined using member initialization as:

```
Circle :: Circle(double r) : radius(r)
{}
```

Structure VS Class

In C++, a structure is same as class except the following differences:

- Members of a class are private by default and members of struct are public by default.

- When deriving a struct from a class/struct, default access-specifier for a base class/struct is public. And when deriving a class, default access specifier is private.
- Classes are usually used for large amounts of data, whereas structs are usually used for smaller amounts of data.
- Classes can be inherited whereas structures not.
- A structure couldn't be null like a class.
- A structure couldn't have a destructor such as a class.
- A structure can't be abstract, a class can.
- Declared events within a class are automatically locked and then they are thread safe, in contrast to the structure type where events can't be locked.

Static Data Members of the class
We can define class member's static using static keyword. When we declare a member of a class as static it means no matter how many objects of the class are created, there is only one copy of the static member.

A static member is shared by all objects of the class. All static data is initialized to zero when the first object is created, if no other initialization is present. We can't put it in the class definition but it can be initialized outside the class as done in the following example by declaring the static variable, using the scope resolution operator:: to identify which class it belongs to.
Let us try the following example to understand the concept of static data members:

```
#include <iostream>
using namespace std;
class Box {
public:
staticintobjectCount;
    // Constructor definition
Box(double l = 2.0, double b = 2.0, double h = 2.0) {
cout<<"Constructor called." <<endl;
length = l;
breadth = b;
height = h;
      // Increase every time object is created
objectCount++;
    }
double Volume() {
return length * breadth * height;
    }
private:
double length;     // Length of a box
double breadth;    // Breadth of a box
double height;     // Height of a box
};
// Initialize static member of class Box
int Box::objectCount = 0;
int main(void) {
  Box Box1(3.3, 1.2, 1.5);    // Declare box1
  Box Box2(8.5, 6.0, 2.0);    // Declare box2

  // Print total number of objects.
cout<< "Total objects: " << Box::objectCount<<endl;
return 0;
}
```

Static Function Members

By declaring a function member as static, you make it independent of any particular object of the class. A static member function can be called even if no objects of the class exist and the static functions are accessed using only the class name and the scope resolution operator ::.

A static member function can only access static data member, other static member functions and any other functions from outside the class.

Static member functions have a class scope and they do not have access to the this pointer of the class. You could use a static member function to determine whether some objects of the class have been created or not.

Array as Data Members

Arrays can be declared as the members of a class. The arrays can be declared as private, public or protected members of the class.

Example: A program to demonstrate the concept of arrays as class members

```
#include<iostream>
using namespace std;
constint size=5;
class student
{
introll_no;
int marks[size];
public:
voidgetdata ();
voidtot_marks ();
} ;
void student :: getdata ()
{
cout<<"\nEnter roll no: ";
Cin>>roll_no;
for(int i=0; i<size; i++)
{
cout<<"Enter marks in subject"<<(i+1)<<": ";
cin>>marks[i] ;
}

void student :: tot_marks() //calculating total marks
{
int total=0;
for(int i=0; i<size; i++)
total+ = marks[i];
cout<<"\n\nTotal marks "<<total;
}
int main()
studentstu;
stu.getdata() ;
stu.tot_marks() ;
return 0;
}
```

String class

Strings are objects that represent sequences of characters.

The standard string class provides support for such objects with an interface similar to that of a standard container of bytes, but adding features specifically designed to operate with strings of single-byte characters.

The string class is an instantiation of the basic_string class template that uses char (i.e., bytes) as its character type, with its default char_traits and allocator types (see basic_string for more info on the template).

Note that this class handles bytes independently of the encoding used: If used to handle sequences of multi-byte or variable-length characters (such as UTF-8), all members of this class (such as length or size), as well as its iterators, will still operate in terms of bytes (not actual encoded characters).

Run time and Compile time polymorphism

Polymorphism means “Poly mean Multiple” + “Morph means Forms” . It is one feature of Object Oriented Paradigm having ability of taking more than one form.

Compile time Polymorphism	Run time Polymorphism
In Compile time Polymorphism, call is resolved by the compiler.	In Run time Polymorphism, call is not resolved by the compiler.
It is also known as Static binding, Early binding and overloading as well.	It is also known as Dynamic binding, Late binding and overriding as well.
Overloading is compile time polymorphism where more than one methods share the same name with different parameters or signature and different return type.	Overriding is run time polymorphism having same method with same parameters or signature, but associated in a class & its subclass.
It is achieved by function overloading and operator overloading.	It is achieved by virtual functions and pointers.
It provides fast execution because known early at compile time.	It provides slow execution as compare to early binding because it is known at runtime.
Compile time polymorphism is less flexible as all things execute at compile time.	Run time polymorphism is more flexible as all things execute at run time.

UNIT III

Overloading in C++

C++ allows you to specify more than one definition for a function name or an operator in the same scope, which is called function overloading and operator overloading respectively.

An overloaded declaration is a declaration that had been declared with the same name as a previously declared declaration in the same scope, except that both declarations have different arguments and obviously different definition (implementation).

When you call an overloaded function or operator, the compiler determines the most appropriate definition to use by comparing the argument types you used to call the function or operator with the parameter types specified in the definitions. The process of selecting the most appropriate overloaded function or operator is called overload resolution.

Function overloading in C++

You can have multiple definitions for the same function name in the same scope. The definition of the function must differ from each other by the types and/or the number of arguments in the argument list. You cannot overload function declarations that differ only by return type.

Following is the example where same function print() is being used to print different data types:

```
#include <iostream>
using namespace std;
classprintData {
public:
void print(int i) {
cout<< "Printing int: " << i <<endl;
   }
void print(double  f) {
cout<< "Printing float: " << f <<endl;
   }

void print(char* c) {
cout<< "Printing character: " << c <<endl;
   }
};
int main(void) {
printDatapd;

  // Call print to print integer
pd.print(5);

  // Call print to print float
pd.print(500.263);

  // Call print to print character
pd.print("Hello C++");
return 0;
}
```

Overloading unary operators

The unary operators operate on a single operand and following are the examples of Unary operators −

The increment (++) and decrement (--) operators.
The unary minus (-) operator.
The logical not (!) operator.
The unary operators operate on the object for which they were called and normally, this operator appears on the left side of the object, as in !obj, -obj, and ++obj but sometime they can be used as postfix as well like obj++ or obj--.

Following example explain how minus (-) operator can be overloaded for prefix as well as postfix usage.

```
#include <iostream>
using namespace std;
class Distance {
private:
int feet;           // 0 to infinite
int inches;          // 0 to 12

public:
    // required constructors
Distance() {
feet = 0;
inches = 0;
    }
Distance(int f, int i) {
feet = f;
inches = i;
    }

    // method to display distance
voiddisplayDistance() {
cout<< "F: " << feet << " I:" << inches <<endl;
    }

    // overloaded minus (-) operator
    Distance operator- () {
feet = -feet;
inches = -inches;
return Distance(feet, inches);
    }
};

int main() {
  Distance D1(11, 10), D2(-5, 11);

  -D1;                  // apply negation
D1.displayDistance();    // display D1
```

```
   -D2;                  // apply negation
D2.displayDistance();    // display D2

return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
F: -11 I:-10
F: 5 I:-11
```

Overloading binary operators

The binary operators take two arguments and following are the examples of Binary operators. You use binary operators very frequently like addition (+) operator, subtraction (-) operator and division (/) operator.

Following example explains how addition (+) operator can be overloaded. Similar way, you can overload subtraction (-) and division (/) operators.

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

class Box {
double length;      // Length of a box
double breadth;     // Breadth of a box
double height;      // Height of a box

public:

doublegetVolume(void) {
return length * breadth * height;
   }

voidsetLength( double len ) {
length = len;
   }

voidsetBreadth( double bre ) {
breadth = bre;
   }

voidsetHeight( double hei ) {
height = hei;
   }
      // Overload + operator to add two Box objects.
   Box operator+(const Box& b) {
      Box box;
box.length = this->length + b.length;
box.breadth = this->breadth + b.breadth;
box.height = this->height + b.height;
return box;
   }
};
// Main function for the program
int main() {
   Box Box1;              // Declare Box1 of type Box
   Box Box2;              // Declare Box2 of type Box
   Box Box3;              // Declare Box3 of type Box
double volume = 0.0;    // Store the volume of a box here

   // box 1 specification
Box1.setLength(6.0);
Box1.setBreadth(7.0);
Box1.setHeight(5.0);

   // box 2 specification
Box2.setLength(12.0);
Box2.setBreadth(13.0);
Box2.setHeight(10.0);

   // volume of box 1
volume = Box1.getVolume();
cout<< "Volume of Box1 : " << volume <<endl;

   // volume of box 2
volume = Box2.getVolume();
cout<< "Volume of Box2 : " << volume <<endl;

   // Add two object as follows:
   Box3 = Box1 + Box2;

   // volume of box 3
volume = Box3.getVolume();
cout<< "Volume of Box3 : " << volume <<endl;

return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
Volume of Box1 : 210
Volume of Box2 : 1560
Volume of Box3 : 5400
```

Data Conversion and Pitfalls of Operators Overloading

Data conversion in C++ includes conversions between basic types and user-defined types, and conversions between different user-defined types.

The assignments between types, whether they are basic types or user-defined types, are handled by the compiler with no effort on our part, provided that the same data type is used on both sides of the equal sign.

The different possible data conversion situations are:

1. Conversion between basic and user defined
a) Conversion from basic to user defined data type
b) Conversion from user-defined data type to basic data type

1. Conversion between basic and user defined
(a) Conversion from basic to user defined data type
Conversion from basic to user defined type is done by using the constructor function with one argument of basic type as follows.

Syntax:
```
classclass_name {
private:
   //….
public:
class_name( data_type) {
     // conversion code
  }
};
```
Here is program that illustrates this conversion.
```
//example
//conversion from basic type to object
#include<iostream>
using namespace std;
classcelsius
{
private:
float temper;
public:
celsius()
  {
temper=0;
  } celsius(float ftmp)
  {
temper=(ftmp-32)* 5/9;
  }
voidshowtemper()
  {
cout<<"Temperature in Celsius: "<<temper;
  }
};
int main()
{
celsiuscel;          //cel is user defined
floatfer;           //fer is basic type
cout<<"\nEnter temperature in Fahrenheit measurement: ";
cin>>fer;
cel=fer;                //convert from basic to user-defined; //eqvt to    cel = celsius(fer);
cel.showtemper();
return 0;
}
```

The output of the program is:

```
Enter temperature in Fahrenheit measurement: -40
Temperature in Celsius: -40
```

(b) Conversion from user-defined type to basic data type

Conversion from user-defined type of basic data type is done by overloading the cast operator of basic type as a member function.

Operator function is defined as an overloaded basic data type which takes no arguments.

Return type of operator is not specified because the cast operator function itself specifies the return type.

Syntax:

```
classclass_name {
    ...
public:
operatordata_type() {
            //Conversion code
        }
};
```

Here is example program to illustrate conversion from user-defined type of basic data type.

```
//conversion from user to basic type
#include<iostream>
using namespace std;
classcelsius
{
private:
float temper;
public:
celsius()
  {
temper=0;
  }
operator float()
  {
floatfer;
fer=temper *9/5 + 32;
return (fer);
  }
voidgettemper()
  {
cout<<"\n Enter Temperature in Celsius:";
cin>>temper;
  }
};
int main()
{
celsiuscel;           //cel is user defined
floatfer;            //fer is basic type
cel.gettemper();
fer=cel;              //convert from user-defined to basic;

   //eqvtto  fer= float(cel);

cout<<"\nTemperature in Fahrenheit measurement: "<<fer;
}
```
The output of the program is:
```
Enter Temperature in Celsius: -40
Temperature in Fahrenheit measurement: -40
```

Restrictions on Operator Overloading

Following are some restrictions to be kept in mind while implementing operator overloading.

- Precedence and Associativity of an operator cannot be changed.
- The numbers of Operands cannot be changed. Unary operator remains unary, binary remains binary etc.
- No new operators can be created, only existing operators can be overloaded.
- Cannot redefine the meaning of a procedure. You cannot change how integers are added.

Inheritance

One of the most important concepts in object-oriented programming is that of inheritance. Inheritance allows us to define a class in terms of another class, which makes it easier to create and maintain an application. This also provides an opportunity to reuse the code functionality and fast implementation time.

When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class. This existing class is called the base class, and the new class is referred to as the derived class.

The idea of inheritance implements the is a relationship. For example, mammal IS-A animal, dog IS-A mammal hence dog IS-A animal as well and so on.

Base & Derived Classes

A class can be derived from more than one classes, which means it can inherit data and functions from multiple base classes. To define a derived class, we use a class derivation list to specify the base class(es). A class derivation list names one or more base classes and has the form:

```
class derived-class: access-specifier base-class
```
Where access-specifier is one of public, protected, or private, and base-class is the name of a previously defined class. If the access-specifier is not used, then it is private by default.

Consider a base class Shape and its derived class Rectangle as follows:
```
#include <iostream>
```

```
using namespace std;
// Base class
class Shape {
public:
voidsetWidth(int w) {
width = w;
    }
voidsetHeight(int h) {
height = h;
    }
protected:
int width;
int height;
};
// Derived class
class Rectangle: public Shape {
public:
intgetArea() {
return (width * height);
    }
};
int main(void) {
  Rectangle Rect;
Rect.setWidth(5);
Rect.setHeight(7);
  // Print the area of the object.
cout<< "Total area: " <<Rect.getArea() <<endl;
return 0;
}
```

Access Modifiers

Access	public	protected	private
Same class	yes	yes	yes
Derived classes	yes	yes	no
Outside classes	yes	no	no

Advantages

- Reduce code redundancy.
- Provides code reusability.
- Reduces source code size and improves code readability.
- Code is easy to manage and divided into parent and child classes.
- Supports code extensibility by overriding the base class functionality within child classes.

Disadvantages

- In Inheritance base class and child classes are tightly coupled. Hence If you change the code of parent class, it will get affects to the all the child classes.
- In class hierarchy many data members remain unused and the memory allocated to them is not utilized. Hence affect performance of your program if you have not implemented inheritance correctly.

Types of Inheritance

1. Single inheritance

In this inheritance, a derived class is created from a single base class.

```
//Base Class
class A
{
public void fooA()
 {
 //TO DO:
 }
}
 //Derived Class
class B : A
{
public void fooB()
 {
 //TO DO:
 }
}
```

2. Multi-level inheritance

In this inheritance, a derived class is created from another derived class.

```
//Base Class
class A
{
public void fooA()
 {
 //TO DO:
 }
}

//Derived Class
class B : A
{
public void fooB()
 {
 //TO DO:
 }
}
 //Derived Class
class C : B
{
public void fooC()
```

```
 {
 //TO DO:
 }
}
```

3. Multiple inheritance

In this inheritance, a derived class is created from more than one base class. This inheritance is not supported by .NET Languages like C#, F# etc.

```
//Base Class
class A
{
public void fooA()
 {
 //TO DO:
 }
}
 //Base Class
class B
{
public void fooB()
 {
 //TO DO:
 }
}
 //Derived Class
class C : A, B
{
public void fooC()
 {
 //TO DO:
 }
}
```

4. Hierarchical inheritance

In this inheritance, more than one derived classes are created from a single base.

```
//Base Class
class A
{
public void fooA()
  {
 //TO DO:
 }
}
 //Derived Class
class B : A
{
public void fooB()
 {
 //TO DO:
 }
}
 //Derived Class
class C : A
{
public void fooC()
 {
 //TO DO:
 }
}
 //Derived Class
class D : C
{
public void fooD()
 {
 //TO DO:
 }
}
 //Derived Class
class E : C
{
public void fooE()
 {
 //TO DO:
 }
}
 //Derived Class
class F : B
{
public void fooF()
 {
 //TO DO:
 }
}
 //Derived Class
class G :B
{
public void fooG()
 {
 //TO DO:
 }
}
```

5. Hybrid inheritance

This is combination of more than one inheritance. Hence, it may be a combination of Multilevel and Multiple inheritances or Hierarchical and Multilevel inheritance or Hierarchical and Multipath inheritance or Hierarchical, Multilevel and Multiple inheritances.

Since .NET Languages like C#, F# etc. does not support multiple and multipath inheritance. Hence hybrid inheritance with a combination of multiple or multipath inheritance is not supported by .NET Languages.

```
//Base Class
class A
{
public void fooA()
 {
 //TO DO:
 }
}
 //Base Class
class F
{
public void fooF()
 {
 //TO DO:
 }
}
 //Derived Class
class B : A, F
{
public void fooB()
 {
 //TO DO:
 }
}
 //Derived Class
class C : A
{
public void fooC()
 {
 //TO DO:
 }
}
 //Derived Class
class D : C
{
public void fooD()
 {
 //TO DO:
 }
}
 //Derived Class
class E : C
{
public void fooE()
 {
 //TO DO:
 }
}
```

Derived class constructors
A constructor plays a vital role in initializing an object. An important note, while using constructors during inheritance, is that, as long as a base class constructor does not take any arguments, the derived class need not have a constructor function. However, if a base class contains a constructor with one or more arguments, then it is mandatory for the derived class to have a constructor and pass the arguments to the base class constructor. When both the derived and base class contains constructors, the base constructor is executed first and then the constructor in the derived class is executed.
In case of multiple inheritance, the base class is constructed in the same order in which they appear in the declaration of the derived class. Similarly, in a multilevel inheritance, the constructor will be executed in the order of inheritance.
The derived class takes the responsibility of supplying the initial values to its base class. The constructor of the derived class receives the entire list of required values as its argument and passes them on to the base constructor in the order in which they are declared in the derived class. A base class constructor is called and executed before executing the statements in the body of the derived class.

Public and Private Inheritance
When deriving a class from a base class, the base class may be inherited through public, protected or private inheritance. The type of inheritance is specified by the access-specifier as explained above.

We hardly use protected or private inheritance, but public inheritance is commonly used. While using different type of inheritance, following rules are applied:

Public Inheritance: When deriving a class from a public base class, public members of the base class become public members of the derived class and protected members of the base class become protected members of the derived class. A base class's private members are never accessible directly from a derived class, but can be accessed through calls to the public and protected members of the base class.

Protected Inheritance: When deriving from a protected base class, public and protected members of the base class become protected members of the derived class.

Private Inheritance: When deriving from a private base class, public and protected members of the base class become private members of the derived class.

Unit IV

Addresses and pointers

Pointers are powerful features of C++ that differentiates it from other programming languages like Java and Python. Pointers are used in C++ program to access the memory and manipulate the address.

Address in C++

Each variable you create in your program is assigned a location in the computer's memory. The value the variable stores is actually stored in the location assigned.To know where the data is stored, C++ has an & operator. The & (reference) operator gives you the address occupied by a variable.Ifvar is a variable then, &var gives the address of that variable.

Example 1: Address in C++

```
#include <iostream>
using namespace std;
int main()
{
int var1 = 3;
int var2 = 24;
int var3 = 17;
cout<<&var1 <<endl;
cout<<&var2 <<endl;
cout<<&var3 <<endl;
}
```

Output

```
0x7fff5fbff8ac
0x7fff5fbff8a8
0x7fff5fbff8a4
```

The 0x in the beginning represents the address is in hexadecimal form.

Pointers Variables

C++ gives you the power to manipulate the data in the computer's memory directly. Pointers variables are variables that points to a specific address in the memory pointed by another variable.

How to declare a pointer?

```
int *p;
   OR,
int* p;
```

The statement above defines a pointer variable p. It holds the memory address

The asterisk is a dereference operator which means pointer to.

Here, pointer p is a pointer to int, i.e., it is pointing to an integer value in the memory address.

Reference operator (&) and Deference operator (*)

Reference operator (&) as discussed above gives the address of a variable.

To get the value stored in the memory address, we use the dereference operator (*).

Example 2: C++ Pointers

```
#include <iostream>
using namespace std;
int main() {
int *pc, c;
  c = 5;
cout<< "Address of c (&c): " <<&c <<endl;
cout<< "Value of c (c): " << c <<endl<<endl;

pc = &c;    // Pointer pc holds the memory address of variable c
cout<< "Address that pointer pc holds (pc): "<< pc <<endl;
cout<< "Content of the address pointer pc holds (*pc): " << *pc <<endl<<endl;

   c = 11;    // The content inside memory address &c is changed from 5 to 11.
cout<< "Address pointer pc holds (pc): " << pc <<endl;
cout<< "Content of the address pointer pc holds (*pc): " << *pc <<endl<<endl;

   *pc = 2;
cout<< "Address of c (&c): " <<&c <<endl;
cout<< "Value of c (c): " << c <<endl<<endl;

return 0;
}
```

Output

```
Address of c (&c): 0x7fff5fbff80c
Value of c (c): 5

Address that pointer pc holds (pc): 0x7fff5fbff80c
Content of the address pointer pc holds (*pc): 5

Address pointer pc holds (pc): 0x7fff5fbff80c
Content of the address pointer pc holds (*pc): 11

Address of c (&c): 0x7fff5fbff80c
Value of c (c): 2
```

Pointer and arrays

Pointers are the variables that hold address. Not only can pointers store address of a single variable, it can also store address of cells of an array.

Example 1: C++ Pointers and Arrays

C++ Program to display address of elements of an array using both array and pointers

```
#include <iostream>
using namespace std;
int main()
{
floatarr[5];
float *ptr;
cout<< "Displaying address using arrays: " <<endl;
for (int i = 0; i < 5; ++i)
   {
cout<< "&arr[" << i << "] = " <<&arr[i] <<endl;
   }
   // ptr = &arr[0]
ptr = arr;

cout<<"\nDisplaying address using pointers: "<<endl;
for (int i = 0; i < 5; ++i)
    {
cout<< "ptr + " << i << " = "<<ptr + i <<endl;
    }
return 0;
}
```

Output

```
Displaying address using arrays:
&arr[0] = 0x7fff5fbff880
&arr[1] = 0x7fff5fbff884
&arr[2] = 0x7fff5fbff888
&arr[3] = 0x7fff5fbff88c
&arr[4] = 0x7fff5fbff890

Displaying address using pointers:
ptr + 0 = 0x7fff5fbff880
ptr + 1 = 0x7fff5fbff884
ptr + 2 = 0x7fff5fbff888
ptr + 3 = 0x7fff5fbff88c
ptr + 4 = 0x7fff5fbff890
```

In the above program, a different pointer ptr is used for displaying the address of array elements arr. But, array elements can be accessed using pointer notation by using same array name arr. For example:

```
intarr[3];
&arr[0] is equivalent to arr
&arr[1] is equivalent to arr + 1
&arr[2] is equivalen to arr + 2
```

Example 2: C++ Pointer and Array

C++ Program to insert and display data entered by using pointer notation.

```
#include <iostream>
using namespace std;
int main() {
floatarr[5];
      // Inserting data using pointer notation
cout<< "Enter 5 numbers: ";
for (int i = 0; i < 5; ++i) {
cin>> *(arr + i) ;
    }
   // Displaying data using pointer notation
cout<< "Displaying data: " <<endl;
for (int i = 0; i < 5; ++i) {
cout<< *(arr + i) <<endl ;
    }
return 0;
}
```

Output

```
Enter 5 numbers: 2.5
3.5
```

```
4.5
5
2
Displaying data:
2.5
3.5
4.5
5
2
```

Pointer and Function pointer

This method used is called passing by value because the actual value is passed.However, there is another way of passing an argument to a function where where the actual value of the argument is not passed. Instead, only the reference to that value is passed.

Example 1: Passing by reference without pointers

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

// Function prototype
void swap(int*, int*);
int main()
{
int a = 1, b = 2;
cout<< "Before swapping" <<endl;
cout<< "a = " << a <<endl;
cout<< "b = " << b <<endl;

swap(&a, &b);

cout<< "\nAfter swapping" <<endl;
cout<< "a = " << a <<endl;
cout<< "b = " << b <<endl;
return 0;
}
void swap(int* n1, int* n2) {
int temp;
temp = *n1;
   *n1 = *n2;
   *n2 = temp;
}
```

Output

```
Before swapping
a = 1
b = 2

After swapping
a = 2
b = 1
```

Memory management: New and Delete

Arrays can be used to store multiple homogenous data but there are serious drawbacks of using arrays.
we should allocate the memory of an array when you declare it but most of the time, the exact memory needed cannot be determined until runtime.
To avoid wastage of memory, you can dynamically allocate memory required during runtime using new and delete operator in C++.
Example 1: C++ Memory Management
C++ Program to store GPA of n number of students and display it where n is the number of students entered by user.

```
#include <iostream>
using namespace std;
class Test
{
private:
intnum;
float *ptr;
public:
Test()
   {
cout<< "Enter total number of students: ";
cin>>num;

ptr = new float[num];

cout<< "Enter GPA of students." <<endl;
for (int i = 0; i <num; ++i)
      {
cout<< "Student" << i + 1 << ": ";
cin>> *(ptr + i);
      }
   }
      ~Test() {
delete[] ptr;
   }

void Display() {
cout<< "\nDisplaying GPA of students." <<endl;
for (int i = 0; i <num; ++i) {
cout<< "Student" << i+1 << " :" << *(ptr + i) <<endl;
      }
   }
};
int main() {
   Test s;
s.Display();
return 0;
}
```

Output:

```
Enter total number of students: 4
Enter GPA of students.
Student1: 3.6
Student2: 3.1
Student3: 3.9
Student4: 2.9

Displaying GPA of students.
Student1 :3.6
Student2 :3.1
Student3 :3.9
Student4 :2.9
```

When the object s is created, the constructor is called which allocates the memory for num floating-point data.
When the object is destroyed, i.e, when the object goes out of scope, destructor is automatically called.

The new Operator
ptr = new float[num];
This expression in the above program returns a pointer to a section of memory just large enough to hold the num number of floating-point data.

The delete Operator
Once the memory is allocated using new operator, it should released back to the operating system.
If the program uses a large amount of memory using new, system may crash because there will be no memory available for the operating system.
The following expression returns memory back to the operating system.
delete [] ptr;
The brackets [] indicates the array has been deleted. If you need to delete a single object then, you don't need to use brackets.
deleteptr;

Pointers to objects
A variable that holds an address value is called a pointer variable or simply pointer.
Pointer can point to objects as well as to simple data types and arrays.
we can use new to create objects while the program is running. new returns a pointer to an unnamed objects.
Example:

```
#include <iostream>
#include <string>
using namespace std;
class student
{
private:
introllno;
string name;
public:
student():rollno(0),name("")
        {}
student(int r, string n): rollno(r),name (n)
```

```
        {}
void get()
        {
cout<<"enter roll no";
cin>>rollno;
cout<<"enter name";
cin>>name;
        }
void print()
        {
cout<<"roll no is "<<rollno;
cout<<"name is "<<name;
        }
};
void main ()
{
student *ps=new student;
        (*ps).get();
        (*ps).print();
deleteps;
}
```

Dangling pointer

A pointer pointing to a memory location that has been deleted (or freed) is called dangling pointer. There are three different ways where Pointer acts as dangling pointer.

De-allocation of memory

// Deallocating a memory pointed by ptr causes

```
// dangling pointer
#include <stdlib.h>
#include  <stdio.h>
int main()
{
int *ptr = (int *)malloc(sizeof(int));

   // After below free call, ptr becomes a
   // dangling pointer
free(ptr);

   // No more a dangling pointer
ptr = NULL;
}
```

Virtual Function

Giving new implementation of base class method into derived class and the calling of this new implemented function with derived class's object is called function overriding.

Giving new implementation of derived class method into base class and the calling of this new implemented function with base class's object is done by making base class function as virtual function.
Virtual function is used in situation, when we need to invoke derived class function using base class pointer. We must declare base class function as virtual using virtual keyword preceding its normal declaration. The base class object must be of pointer type so that we can dynamically replace the address of base class function with derived class function. This is how we can achieve "‘untime Polymorphism”. If we doesn't use virtual keyword in base class, base class pointer will always execute function defined in base class.
Example of virtual function

```
    #include<iostream.h>
    #include<conio.h>
classBaseClass
    {

public:
virtual void Display()
        {
cout<<"\n\tThis is Display() method of Base Class";
        }

void Show()
        {
cout<<"\n\tThis is Show() method of Base Class";
        }
    };
Class Derived Class : public BaseClass
    {
public:
void Display()
        {
cout<<"\n\tThis is Display() method of Derived Class";
        }

void Show()
        {
cout<<"\n\tThis is Show() method of Derived Class";
        }
    };
void main()
    {
Derived Class D;
BaseClass *B;       //Creating Base Class Pointer
        B = new BaseClass;
        B->Display();       //This will invoke Display() method of Base Class
        B->Show();          //This will invoke Show() method of Base Class
        B=&D;
        B->Display();       //This will invoke Display() method of Derived Class
                           //bcozDisplay() method is virtual in Base Class
```

```
            B->Show();            //This will invoke Show() method of Base Class
                                  //bcozShow() method is not virtual in Base Class

      }
```

Output :

```
            This is Display() method of Base Class
            This is Show() method of Base Class
            This is Display() method of Derived Class
            This is Show() method of Base Class
```

Friend Function

Private members are accessed only within the class they are declared. Friend function is used to access the private and protected members of different classes. It works as bridge between classes. Friend function must be declared with friend keyword. It must be declare in all the classes from which we need to access private or protected members. It will be defined outside the class without specifying the class name. Friend function will be invoked like normal function, without any object.

Example of friend function

```
      #include<iostream.h>
      classRectangleTwo;
      classRectangleOne
      {
            int L,B;
            public:
            RectangleOne(intl,int b)
            {
            L = l;
            B = b;
            }

            friend void Sum(RectangleOne, RectangleTwo);
      };
      Class RectangleTwo
      {
            int L,B;
            public:
            Rectangle Two(intl,int b)
            {
            L = l;
            B = b;
            }

            friend void Sum(Rectangle One, Rectangle Two);
      };
      void Sum(RectangleOne R1,RectangleTwo R2)
      {
            cout<<"\n\t\tLength\tBreadth";
            cout<<"\n Rectangle 1 : "<<R1.L<<"\t "<<R1.B;
            cout<<"\n Rectangle 2  :  "<<R2.L<<"\t  "<<R2.B;
```

```
        cout<<"\n..................................";
        cout<<"\n\tSum  :  "<<R1.L+R2.L<<"\t  "<<R1.B+R2.B;
        cout<<"\n..................................";
    }
    void main()
    {
        RectangleOneRec1(5,3);
        RectangleTwoRec2(2,6);
        Sum(Rec1,Rec2);

    }

    Output :
    Sum :     7        9
```

Static Functions

Static member functions have a class scope and they do not have access to the 'this' pointer of the class. When a member is declared as static, a static member of class, it has only one data for the entire class even though there are many objects created for the class. The main usage of static function is when the programmer wants to have a function which is accessible even when the class is not instantiated. Static function is defined by using the keyword static before the member function that is to be declared as static function.

Syntax

```
staticreturn_data_typefunction_name()
//Static function defined with keyword static
{
statement1;
//Statements for execution inside static function
statement2;
...
...
}
```

Friend Class

A friend class can access all the private and protected members of other class.In order to access the private and protected members of a class into friend class we must pass on object of a class to the member functions of friend class.

Example of C++ friend class:

```
    #include<iostream.h>
    class Rectangle
    {
        int L,B;
        public:
        Rectangle()
        {
            L=10;
            B=20;
        }
        friend class Square;      //Statement 1
    };
    class Square
    {
        int S;
        public:
        Square()
        {
            S=5;
        }
        void Display(Rectangle Rect)
        {
            cout<<"\n\n\tLength : "<<Rect.L;
            cout<<"\n\n\tBreadth : "<<Rect.B;
            cout<<"\n\n\tSide : "<<S;
        }
    };
    void main()
    {
        Rectangle R;
        Square S;
        S.Display(R);     //Statement 2
    }

    Output :
        Length : 10
        Breadth : 20
        Side : 5
```

In the above example, we have created two classes Rectangle and Square. Using statement 1 we have made Square class, a friend class of Rectangle class. In order to access the private and protected members of Rectangle class into Square class we must explicitly pass an object of Rectangle class to the member functions of Square class as shown in statement 2.This is similar to passing an object as function argument but the difference is, an object (R) we are passing as argument is of different class (Rectangle) and the calling object is of different class (Square).

Copy initialization

Consider the following line of code:

int x = 5;

This statement uses copy initialization to initialize newly created integer variable x to the value of 5.

Copy initialization for classes:

```
#include <iostream>
class Fraction
{
private:
intm_numerator;
intm_denominator;
public:
  // Default constructor
Fraction(int numerator=0, int denominator=1) :
m_numerator(numerator), m_denominator(denominator)
   {
assert(denominator != 0);
   }
friendstd::ostream& operator<<(std::ostream& out, const Fraction &f1);
};
std::ostream& operator<<(std::ostream& out, const Fraction &f1)
{
        out<< f1.m_numerator << "/" << f1.m_denominator;
        return out;
}
int main()
{
   Fraction six = Fraction(6);
std::cout<< six;
return 0;
}
```

output:

```
6/1
```

This form of copy initialization is evaluated the same way as the following:

```
        Fraction six(Fraction(6));
```

This Pointer

C++ provides a keyword 'this', which represents the current object and passed as a hidden argument to all member functions. The this pointer is a constant pointer that holds the memory address of the current object. The this pointer is not available in static member functions as static member functions can be called without any object. static member functions can be called with class name.

Example of this pointer

```
        #include<iostream.h>
        #include<conio.h>
        #include<string.h>
        class Student
        {
                int Roll;
                char Name[25];
                float Marks;

                public:
                Student(intR,floatMks,char Nm[])        //Constructor 1
                {
                        Roll = R;
                        strcpy(Name,Nm);
                        Marks = Mks;
                }
                Student(char Name[],float Marks,int Roll)     //Constructor 2
                {
                        Roll = Roll;
                        strcpy(Name,Name);
                        Marks = Marks;
                }
                Student(intRoll,char Name[],float Marks)      //Constructor 3
                {
                        this->Roll = Roll;
                        strcpy(this->Name,Name);
                        this->Marks = Marks;
                }
                void Display()
                {
                        cout<<"\n\tRoll : "<<Roll;
                        cout<<"\n\tName : "<<Name;
                        cout<<"\n\tMarks : "<<Marks;
                }
        };
        void main()
        {
                Student S1(1,89.63,"Sumit");
                Student S2("Kumar",78.53,2);
                Student S3(3,"Gaurav",68.94);

                cout<<"\n\n\tDetails of Student 1 : ";
                S1.Display();

                cout<<"\n\n\tDetails of Student 2 : ";
                S2.Display();

                cout<<"\n\n\tDetails of Student 3 : ";
                S3.Display();

        }
```

Output :

```
Details of Student 1 :
Roll : 1
Name :Sumit
Marks : 89.63

Details of Student 2 :
Roll : 31883
Name : ?&;6•#?#?6•#N$?%_5$?
Marks : 1.07643e+24

Details of Student 3 :
Roll : 3
Name :Gaurav
Marks : 68.94
```

In constructor 1,variables declared in argument list different from variables declared as class data members. When compiler doesn't find Roll, Name, Marks as local variable, then, it will find Roll, Name, Marks in class scope and assign values to them.
But Constructor 2 will not initialize class data members. When we pass values to constructor 2, it will initialize values to itself local variables because variables declared in argument list and variable declared as data members are of same name.
In this situation, we use 'this' pointer to differentiate local variable and class data members as shown in constructor 3.

Dynamic type information

In C++, RTTI (Run-time type information) is a mechanism that exposes information about an object's data type at runtime and is available only for the classes which have at least one virtual function. It allows the type of an object to be determined during program execution
For example, dynamic_cast uses 'TTI and following program fails with error "cannot dynamic_cast `b' (of type `class B*') to type `class D*' (source type is not polymorphic) " because there is no virtual function in the base class B.

```
// Dynamic/Run Time Type Identification
#include<iostream>
using namespace std;
class B
        {
        virtual void fun() {}
        };
class D:
public B
         {
        };

int main()
{
   B *b = new D;
   D *d = dynamic_cast<D*>(b);
if(d != NULL)
cout<< "works";
else
cout<< "cannot cast B* to D*";
getchar();
return 0;
}
```

Output:
```
works
```

UNIT V

Streams classes

We are using the iostream standard library, which provides cin and cout methods for reading from standard input and writing to standard output respectively.

Description	**Data Type**
This data type represents the output file stream and is used to create files and to write information to files.	ofstream
This data type represents the input file stream and is used to read information from files.	ifstream
This data type represents the file stream generally, and has the capabilities of both ofstream and ifstream which means it can create files, write information to files, and read information from files.	fstream

Opening a File

A file must be opened before you can read from it or write to it. Either the ofstream or fstream object may be used to open a file for writing andifstream object is used to open a file for reading purpose only.

Following is the standard syntax for open() function, which is a member of fstream, ifstream, and ofstream objects.

Closing a File

When a C++ program terminates it automatically closes flushes all the streams, release all the allocated memory and close all the opened files. But it is always a good practice that a programmer should close all the opened files before program termination.

Following is the standard syntax for close() function, which is a member of fstream, ifstream, and ofstream objects.

Writing to a File

While doing C++ programming, you write information to a file from your program using the stream insertion operator (<<) just as you use that operator to output information to the screen. The only difference is that you use an ofstream or fstream object instead of the cout object.

Reading from a File

You read information from a file into your program using the stream extraction operator (>>) just as you use that operator to input information from the keyboard. The only difference is that you use an ifstream or fstream object instead of the cin object.

Read & Write Example

Following is the C++ program which opens a file in reading and writing mode. After writing information inputted by the user to a file named afile.dat, the program reads information from the file and outputs it onto the screen:

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

int main () {

char data[100];

   // open a file in write mode.
ofstreamoutfile;
outfile.open("afile.dat");

cout<< "Writing to the file" <<endl;
cout<< "Enter your name: ";
cin.getline(data, 100);

   // write inputted data into the file.
outfile<< data <<endl;

cout<< "Enter your age: ";
cin>> data;
cin.ignore();

   // again write inputted data into the file.
outfile<< data <<endl;

   // close the opened file.
outfile.close();

   // open a file in read mode.
ifstreaminfile;
infile.open("afile.dat");

cout<< "Reading from the file" <<endl;
infile>> data;

   // write the data at the screen.
cout<< data <<endl;

   // again read the data from the file and display it.
infile>> data;
cout<< data <<endl;

   // close the opened file.
infile.close();

return 0;
}
```

Stream Errors
The Standard Error Stream (cerr)
The predefined object cerr is an instance of ostream class. The cerr object is said to be attached to the standard error device, which is also a display screen but the object cerr is un-buffered and each stream insertion to cerr causes its output to appear immediately.
The cerr is also used in conjunction with the stream insertion operator as shown in the following example.

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

int main() {
charstr[] = "Unable to read ... ";

cerr<< "Error message : " <<str<<endl;
}
```

When the above code is compiled and executed, it produces the following result −

```
Error message : Unable to read....
```

Disk File I/O with streams
C++ I/O occurs in streams, which are sequences of bytes. If bytes flow from a device like a keyboard, a disk drive, or a network connection etc. to main memory, this is called input operation and if bytes flow from main memory to a device like a display screen, a printer, a disk drive, or a network connection, etc., this is called output operation.
I/O Library Header Files
There are following header files −
1<iostream>
This file defines the cin, cout, cerr and clog objects, which correspond to the standard input stream, the standard output stream, the un-buffered standard error stream and the buffered standard error stream, respectively.
2<iomanip>
This file declares services useful for performing formatted I/O with so-called parameterized stream manipulators, such as setw and setprecision.
3<fstream>
This file declares services for user-controlled file processing.

The Standard Output Stream (cout)
The predefined object cout is an instance of ostream class. The cout object is said to be "connected to" the standard output device, which usually is the display screen. The cout is used in conjunction with the stream insertion operator, which is written as << which are two less than signs as shown in the following example.

```
#include <iostream>

using namespace std;

int main() {
charstr[] = "Hello C++";

cout<< "Value of str is : " <<str<<endl;
}
```

```
Output –
Value of stris : Hello C++
```

The Standard Input Stream (cin)

The predefined object cin is an instance of istream class. The cin object is said to be attached to the standard input device, which usually is the keyboard. The cin is used in conjunction with the stream extraction operator, which is written as >> which are two greater than signs as shown in the following example.

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

int main() {
char name[50];

cout<< "Please enter your name: ";
cin>> name;
cout<< "Your name is: " << name <<endl;
}
```

```
Output –
Please enter your name: cplusplus
Your name is: cplusplus
```

File Pointers

Both istream and ostream provide member functions for repositioning the file-position pointer. These member functions are seekg ("seek get") for istream and seekp ("seek put") for ostream.

The argument to seekg and seekp normally is a long integer. A second argument can be specified to indicate the seek direction. The seek direction can be ios::beg (the default) for positioning relative to the beginning of a stream, ios::cur for positioning relative to the current position in a stream or ios::end for positioning relative to the end of a stream.

The file-position pointer is an integer value that specifies the location in the file as a number of bytes from the file's starting location. Some examples of positioning the "get" file-position pointer are:

```
// position to the nth byte of fileObject (assumes ios::beg)
fileObject.seekg( n );

// position n bytes forward in fileObject
fileObject.seekg( n, ios::cur );

// position n bytes back from end of fileObject
fileObject.seekg( n, ios::end );

// position at end of fileObject
fileObject.seekg( 0, ios::end );
```

C++ Error Handling Functions

There are several error handling functions supported by class ios that help you read and process the status recorded in a file stream.

Some error handling functions and their meaning :

- int bad()-Returns a non-zero value if an invalid operation is attempted or any unrecoverable error has occurred. However, if it is zero (false value), it may be possible to recover from any other error reported and continue operations.
- inteof()-Returns non-zero (true value) if end-of-file is encountered while reading; otherwise returns zero (false value).
- int fail()-Returns non-zero (true) when an input or output operation has failed.
- int good()-Returns non-zero (true) if no error has occurred. This means, all the above functions are false. For example, if fin.good() is true, everything is okay with the stream named as fin and we can proceed to perform I/O operations. When it returns zero, no further operations can be carried out.
- clear()- Resets the error state so that further operations can be attempted.

Overloading the extraction and insertion operators

C++ is able to input and output the built-in data types using the stream extraction operator >> and the stream insertion operator <<. The stream insertion and stream extraction operators also can be overloaded to perform input and output for user-defined types like an object.

Here, it is important to make operator overloading function a friend of the class because it would be called without creating an object.

Following example explains how extraction operator >> and insertion operator <<.

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

class Distance {
private:
int feet;            // 0 to infinite
int inches;           // 0 to 12
public:
    // required constructors
Distance(){
feet = 0;
inches = 0;
    }
Distance(int f, int i){
feet = f;
inches = i;
    }
friendostream&operator<<( ostream&output,
const Distance &D ) {
output<< "F : " <<D.feet<< " I : " <<D.inches;
return output;
    }
friendistream&operator>>( istream&input, Distance &D ) {
input>>D.feet>>D.inches;
return input;
    }
};

int main() {
  Distance D1(11, 10), D2(5, 11), D3;

cout<< "Enter the value of object : " <<endl;
cin>> D3;
cout<< "First Distance : " << D1 <<endl;
cout<< "Second Distance :" << D2 <<endl;
cout<< "Third Distance :" << D3 <<endl;

return 0;
}
```

Command line arguments in C/C++

The most important function of C/C++ is main() function. It is mostly defined with a return type of int and without parameters :

int main() { /* ... */ }

Command-line arguments are given after the name of the program in command-line shell of Operating Systems.

To pass command line arguments, we typically define main() with two arguments : first argument is the number of command line arguments and second is list of command-line arguments.

int main(intargc, char *argv[]) { /* ... */ }

or

int main(intargc, char **argv[]) { /* ... */ }

argc (ARGument Count) is int and stores number of command-line arguments passed by the user including the name of the program. So if we pass a value to a program, value of argc would be 2 (one for argument and one for program name)

The value of argc should be non negative.

argv(ARGument Vector) is array of character pointers listing all the arguments.

If argc is greater than zero,the array elements from argv[0] to argv[argc-1] will contain pointers to strings.

Argv[0] is the name of the program , After that till argv[argc-1] every element is command -line arguments.

```
// program mainreturn.cpp
#include <iostream>
using namespace std;

int main(intargc, char** argv)
{
cout<< "You have entered " <<argc
<<" arguments:" << "\n";

for (int i = 0; i <argc; ++i)
cout<<argv[i] << "\n";
return 0;
}
```

Input:
```
$ g++ mainreturn.cpp -o main
$ ./main geeks for geeks
```

Output:
```
You have entered 4 arguments:
./main
geeks
for
geeks
```

Templates
Templates are the foundation of generic programming, which involves writing code in a way that is independent of any particular type.

A template is a blueprint or formula for creating a generic class or a function. The library containers like iterators and algorithms are examples of generic programming and have been developed using template concept.

There is a single definition of each container, such as vector, but we can define many different kinds of vectors for example, vector <int> or vector <string>.

You can use templates to define functions as well as classes, let us see how do they work:

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

```
template<class type> ret-type func-name(parameter list) {
  // body of function
}
```
Here, type is a placeholder name for a data type used by the function. This name can be used within the function definition.

The following is the example of a function template that returns the maximum of two values:
```
#include <iostream>
#include <string>

using namespace std;

template<typename T>
inline T const& Max (T const& a, T const& b) {
return a < b ? b:a;
}

int main () {

int i = 39;
int j = 20;
cout<< "Max(i, j): " << Max(i, j) <<endl;
double f1 = 13.5;
double f2 = 20.7;
cout<< "Max(f1, f2): " << Max(f1, f2) <<endl;

string s1 = "Hello";
string s2 = "World";
cout<< "Max(s1, s2): " << Max(s1, s2) <<endl;

return 0;
}
```
If we compile and run above code, this would produce the following result:

```
Max(i, j): 39
Max(f1, f2): 20.7
Max(s1, s2): World
```
Class Template

Just as we can define function templates, we can also define class templates. The general form of a generic class declaration is shown here:

```
template<class type> class class-name {
   .
   .
   .
}
```
Here, type is the placeholder type name, which will be specified when a class is instantiated. You can define more than one generic data type by using a comma-separated list.

Following is the example to define class Stack<> and implement generic methods to push and pop the elements from the stack:

```
#include <iostream>
#include <vector>
#include <cstdlib>
#include <string>
#include <stdexcept>

using namespace std;

template<class T>
class Stack {
private:
vector<T>elems;     // elements

public:
void push(T const&); // push element
void pop();             // pop element
    T top() const;          // return top element
bool empty() const{      // return true if empty.
returnelems.empty();
      }
};

template<class T>
void Stack<T>::push (T const&elem) {
   // append copy of passed element
elems.push_back(elem);
}

template<class T>
void Stack<T>::pop () {
if (elems.empty()) {
throwout_of_range("Stack<>::pop(): empty stack");
   }

   // remove last element
elems.pop_back();
}

template<class T>
T Stack<T>::top () const {
if (elems.empty()) {
throwout_of_range("Stack<>::top(): empty stack");
   }

   // return copy of last element
returnelems.back();
}

int main() {
try {
      Stack<int>intStack; // stack of ints
      Stack<string>stringStack;    // stack of strings

      // manipulate int stack
intStack.push(7);
cout<<intStack.top() <<endl;

      // manipulate string stack
stringStack.push("hello");
cout<<stringStack.top() <<std::endl;
stringStack.pop();
stringStack.pop();
}catch (exception const& ex) {
cerr<< "Exception: " <<ex.what() <<endl;
return -1;
   }
}
```

Exception handling

An exception is a problem that arises during the execution of a program. A C++ exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero.

Exceptions provide a way to transfer control from one part of a program to another. C++ exception handling is built upon three keywords: try, catch, and throw.

throw: A program throws an exception when a problem shows up. This is done using a throw keyword.

catch: A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The catch keyword indicates the catching of an exception.

try: A try block identifies a block of code for which particular exceptions will be activated. It's followed by one or more catch blocks.

Assuming a block will raise an exception, a method catches an exception using a combination of the try and catch keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch looks like the following:

```
try
{
  // protected code
}catch( ExceptionName e1 )
{
  // catch block
}catch( ExceptionName e2 )
{
  // catch block
}catch( ExceptionNameeN )
{
  // catch block
}
```

You can list down multiple catch statements to catch different type of exceptions in case your try block raises more than one exception in different situations.

Throwing Exceptions

Exceptions can be thrown anywhere within a code block using throw statements. The operand of the throw statements determines a type for the exception and can be any expression and the type of the result of the expression determines the type of exception thrown.

Following is an example of throwing an exception when dividing by zero condition occurs:

```
double division(int a, int b) {
if( b == 0 ) {
throw "Division by zero condition!";
   }
return (a/b);
}
```

Catching Exceptions

The catch block following the try block catches any exception. You can specify what type of exception you want to catch and this is determined by the exception declaration that appears in parentheses following the keyword catch.

```
try {
  // protected code
}catch( ExceptionName e ) {
  // code to handle ExceptionName exception
}
```

Above code will catch an exception of ExceptionName type. If you want to specify that a catch block should handle any type of exception that is thrown in a try block, you must put an ellipsis, ..., between the parentheses enclosing the exception declaration as follows:

```
try {
  // protected code
}catch(...) {
  // code to handle any exception
}
```

The following is an example, which throws a division by zero exception and we catch it in catch block.

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

double division(int a, int b) {
if( b == 0 ) {
throw "Division by zero condition!";
  }
return (a/b);
}

int main () {
int x = 50;
int y = 0;
double z = 0;

try {
    z = division(x, y);
cout<< z <<endl;
}catch (const char* msg) {
cerr<<msg<<endl;
  }

return 0;
}
```

Because we are raising an exception of type const char*, so while catching this exception, we have to use const char* in catch block. If we compile and run above code, this would produce the following result: