BCS 201: OBJECT ORIENTED PROGRAMMING

Module-1 **(10 Lectures)**

Introduction to object oriented programming, user-defined types, polymorphism, and encapsulation. Getting started with C++ syntax, data-type, various, functions, exceptions and statement, namespaces and exceptions, operators, flow control, functions, recursion. Arrays and pointers, structures.

Module – II **(10 Lectures)**

Abstraction mechanisms: Classes, private, public construction, member functions, static members, references etc. class hierarchy, derived classes.

Inheritance: simple inheritance, polymorphism, object slicing, base initialization virtual functions.

Module – III **(12 Lectures)**

Prototypes, linkages, operator overloading, ambiguity, friends, member operators, operator function, I/O operator etc. Memory management: new delete, object copying copy constructors, assignment operator, this input/output. Exception handling: Exceptions and derived classes, function exception declarations, Unexpected exceptions, Exceptions when handling exceptions, resource capture and release etc.

Module – IV **(08 Lectures)**

Templates and standard Template library: template classes, declaration, template functions, namespaces, string, iterates hashes, iostreams and other type.

Design using C++ design and development, design and programming, role of classes.

Introduction to C++

Starting with C++
How C++ evolved from C?
OOP vs. procedure-oriented programming
Features of C++
The basic anatomy of a C++ program
Compiling, linking and running a C++ program

Note:

The history of C++ begins with C. C++ is built upon the foundation of C. Thus, C++ is a superset of C. C++ expanded and enhanced the C language to support object-oriented programming. C++ also added several other improvements to the C language, including an extended set of library routines. However, much of the spirit and flavor of C++ is directly inherited from C. Therefore, to fully understand and appreciate C++, you need to understand the "how and why" behind C.

C++ was invented by Bjarne Stroustrup in 1979, at Bell Laboratories in Murray Hill, New Jersey. He initially called the new language "C with Classes" However in 1983 the name was changed to C++ Stroustrup built C++ on the foundation of C including all of C's features attributes and benefits. Most of the features that Stroustrup added to C were designed to support object-oriented programming. In essence, C++ is the object-oriented version of C. By building upon the foundation of C, Stroustrup provided a smooth migration path to OOP. Instead of having to learn an entirely new language, a C programmer needed to learn only a few new features before reaping the benefits of the object-oriented methodology.

Procedure-Oriented Programming Vs. OOP

In the procedure oriented approach, the problem is viewed as the sequence of things to be done such as reading, calculating and printing such as C, Pascal, fortran etc. The primary focus is on functions. A typical structure for procedural programming is shown in fig.1.1. The technique of hierarchical decomposition has been used to specify the tasks to be completed for solving a problem.

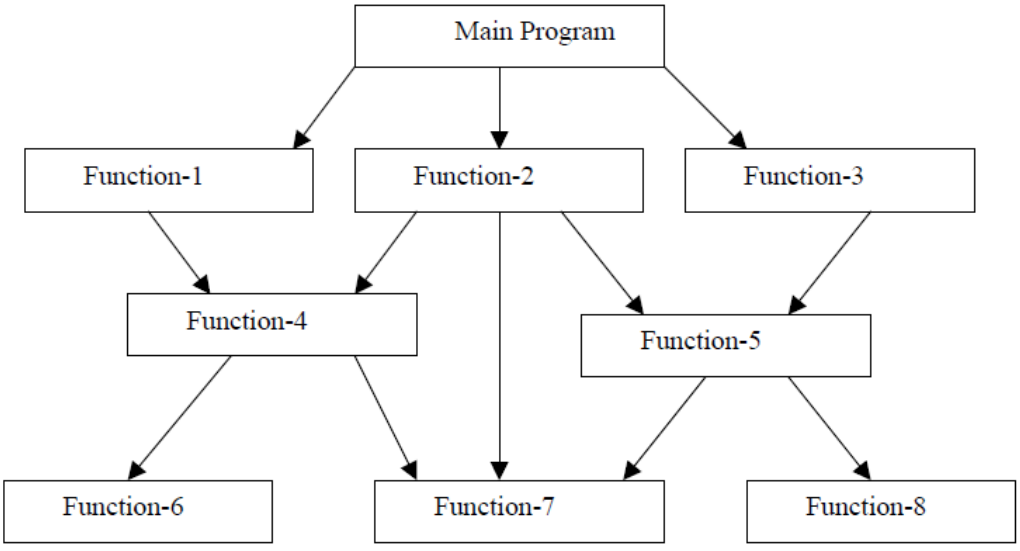


Fig. 1.1 Structure of procedural oriented programs

Procedure oriented programming basically consists of writing a list of instructions for the computer to follow and organizing these instructions into groups known as

functions. We normally use flowcharts to organize these actions and represent the flow of control from one action to another.

In a multi-function program, many important data items are placed as global so that they may be accessed by all the functions. Each function may have its own local data. Global data are more vulnerable to an inadvertent change by a function. In a large program it is very difficult to identify what data is used by which function. In case we need to revise an external data structure, we also need to revise all functions that access the data. This provides an opportunity for bugs to creep in.

Another serious drawback with the procedural approach is that we do not model real world problems very well. This is because functions are action-oriented and do not really corresponding to the element of the problem.

Some Characteristics exhibited by procedure-oriented programming are:

- Emphasis is on doing things (algorithms).
- Large programs are divided into smaller programs known as functions.
- Most of the functions share global data.
- Data move openly around the system from function to function.
- Functions transform data from one form to another.
- Employs top-down approach in program design.

Object-Oriented Programming (OOP)

The major motivating factor in the invention of object-oriented approach is to remove some of the flaws encountered in the procedural approach. OOP treats data as a critical element in the program development and does not allow it to flow freely around the system. It ties data more closely to the function that operate on it, and protects it from accidental modification from outside function. OOP allows decomposition of a problem into a number of entities called objects and then builds data and function around these objects. The organization of data and function in object-oriented programs is shown in fig.1.2. The data of an object can be accessed only by the function associated with that object. However, function of one object can access the function of other objects.

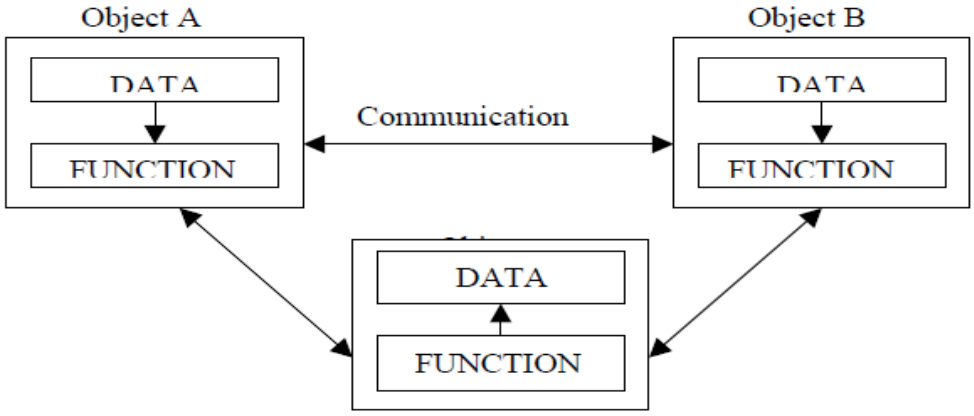


Fig. 1.2 Organization of data and function in OOP

Some Characteristics of Object Oriented Programming

- Emphasis is on data rather than procedure.
- Programs are divided into what are known as objects.
- Data structures are designed such that they characterize the objects.
- Functions that operate on the data of an object are ties together in the data structure.
- Data is hidden and cannot be accessed by external function.

- Objects may communicate with each other through function.
- New data and functions can be easily added whenever necessary.
- Follows bottom up approach in program design.

To support the principles of object-oriented programming, all OOP languages, including C++, have the following characteristics

- Objects
- Classes
- Data abstraction and encapsulation
- Inheritance
- Polymorphism
- Dynamic binding
- Message passing

Let's examine each.

Objects

Objects are the basic run time entities in an object-oriented system. They may represent a person, a place, a bank account, a table of data or any item that the program has to handle. They may also represent user-defined data such as vectors, time and lists. Programming problem is analyzed in term of objects and the nature of communication between them. Program objects should be chosen such that they match closely with the real-world objects. Objects take up space in the memory and have an associated address like a record in Pascal, or a structure in c.
When a program is executed, the objects interact by sending messages to one another. For example, if "customer" and "account" are to object in a program, then the customer object may send a message to the count object requesting for the bank balance. Each object contain data, and code to manipulate data. Objects can interact without having to know details of each other's data or code. It is sufficient to know the type of message accepted, and the type of response returned by the objects.

Classes

As mentioned above the objects contain data, and code to manipulate that data. The entire set of data and code of an object can be made a user-defined data type with the help of class. In fact, objects are variables of the type class. Once a class has been defined, we can create any number of objects belonging to that class. Each object is associated with the data of type class with which they are created. A class is thus a collection of objects similar types. For examples, guava, mango and orange are members of the class fruit. Classes are user-defined that types and behave like the built-in types of a programming language. The syntax used to create an object is not different then the syntax used to create an integer object in C. If fruit has been defines as a class, then the statement
Fruit Mango; will create an object mango belonging to the class fruit.

Data Abstraction and Encapsulation

Encapsulation is a programming mechanism that binds together code and the data it manipulates, and that keeps both safe from outside interference and misuse. In an object-oriented language, code and data can be bound together in such a way that a self-contained black box is created. Within the box are all necessary data and code.

When code and data are linked together in this fashion, an object is created. In other words, an object is the device that supports encapsulation.
Abstraction refers to the act of representing essential features without including the background details or explanation. Classes use the concept of abstraction and are defined as a list of abstract attributes such as size, wait, and cost, and function operate on these attributes. They encapsulate all the essential properties of the object that are to be created.
The attributes are some time called data members because they hold information. The functions that operate on these data are sometimes called methods or member function.

Polymorphism

Polymorphism (from Greek meaning "many forms") is the quality that allows one interface to access a general class of actions. A simple example of polymorphism is found in the steering wheel of an automobile. The steering wheel (the interface) is the same no matter what type of actual steering mechanism is used. That is, the steering wheel works the same whether your car has manual steering, power steering, or rack-and-pinion steering. Thus, turning the steering wheel left causes the car to go left no matter what type of steering is used. The benefit of the uniform interface is, of course, that once you know how to operate the steering wheel, you can drive any type of car.
The same principle can also apply to programming. For example, consider a stack (which is a first-in, last- out list). You might have a program that requires three different types of stacks. One stack is used for integer values, one for floating-point values, and one for characters. In this case, the algorithm that implements each stack is the same, even though the data being stored differs.
In a non–object-oriented language, you would be required to create three different sets of stack routines, with each set using different names. However, because of polymorphism, in C++ you can create one general set of stack routines that works for all three situations. This way, once you know how to use one stack, you can use them all.

Inheritance

Inheritance is the process by which one object can acquire the properties of another object. This is important because it supports the concept of hierarchical classification. If you think about it, most knowledge is made manageable by hierarchical (that is, top-down) classifications. For example, a Red Delicious apple is part of the classification apple, which in turn is part of the fruit class, which is under the larger class food. That is, the food class possesses certain qualities (edible, nutritious, and so on) which also, logically, apply to its subclass, fruit. In addition to these qualities, the fruit class has specific characteristics (juicy, sweet, and so on) that distinguish it from other food. The apple class defines those qualities specific to an apple (grows on trees, not tropical, and so on). A Red Delicious apple would, in turn, inherit all the qualities of all preceding classes and would define only those qualities that make it unique. Without the use of hierarchies, each object would have to explicitly define all of its characteristics. Using inheritance, an object need only define those qualities that make it unique within its class. It can inherit its general attributes from its parent. Thus, it is the inheritance mechanism that makes it possible for one object to be a specific instance of a more general case.

Dynamic Binding

Binding refers to the linking of a procedure call to the code to be executed in response to the call. Dynamic binding means that the code associated with a given procedure call is not known until the time of the call at run time. It is associated with polymorphism and inheritance. A function call associated with a polymorphic reference depends on the dynamic type of that reference.
Consider the procedure "draw" by inheritance, every object will have this procedure. Its algorithm is, however, unique to each object and so the draw procedure will be redefined in each class that defines the object. At run-time, the code matching the object under current reference will be called.

Message Passing

An object-oriented program consists of a set of objects that communicate with each other. Objects communicate with one another by sending and receiving information. A Message for an object is a request for execution of a procedure, and therefore will invoke a function (procedure) in the receiving object that generates the desired results. Message passing involves specifying the name of object, the name of the function (message) and the information to be sent.

Application of OOP

OOP has become one of the programming buzzwords today. There appears to be a great deal of excitement and interest among software engineers in using OOP. Applications of OOP are beginning to gain importance in many areas. The most popular application of object-oriented programming, up to now, has been in the area of user interface design such as window. Hundreds of windowing systems have been developed, using the OOP techniques.
Real-business system are often much more complex and contain many more objects with complicated attributes and method. OOP is useful in these types of application because it can simplify a complex problem. The promising areas of application of OOP include:

- Real-time system
- Simulation and modeling
- Object-oriented data bases
- Hypertext, Hypermedia, and expertext
- AI and expert systems
- Neural networks and parallel programming
- Decision support and office automation systems
- CIM/CAM/CAD systems

An Example of C++ Program

Here is an example of a complete C++ program:

```
// The C++ compiler ignores comments which start with
// double slashes like this, up to the end of the line.

/* Comments can also be written starting with a slash
 followed by a star, and ending with a star followed by
 a slash. As you can see, comments written in this way
```

```
 can span more than one line. */

/* Programs should ALWAYS include plenty of comments! */

/* Author: Rob Miller and William Knottenbelt
 Program last changed: 30th September 2001 */

/* This program prompts the user for the current year, the user's
current age, and another year. It then calculates the age
that the user was or will be in the second year entered. */

#include <iostream>

using namespace std;
{
        cout<<" c++ is better than c \n";
        return 0;
}
```

Input Operator cin :-

The identifier cin is a predefined object in c++ that corresponds to the standard input stream. Here this stream represents keyboard. Syntax:- cin>>variable; The operator >> is known as extraction or get from operator & assigns it to the variable on its right.

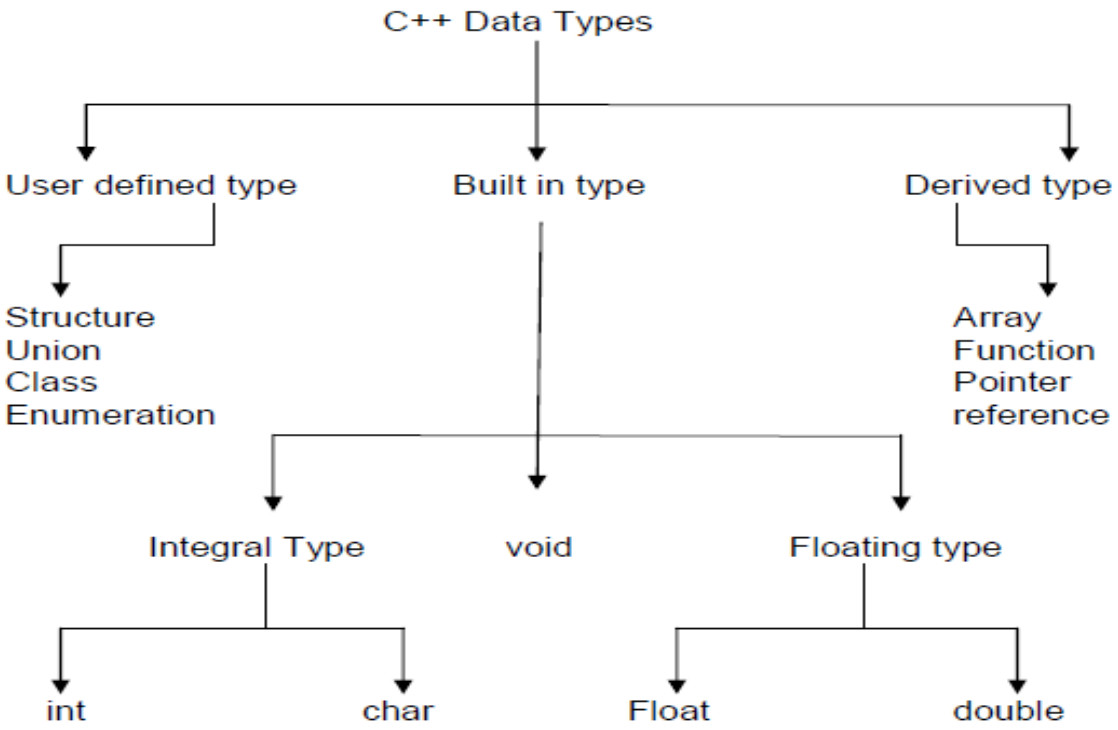
2. Output operator cout :-

The identifier cout is predefined object that represents the standard output stream in c++. Here standard output stream represents the screen. Syntax:- cout<<string; The operator << is called the insertion or put to operator. It inserts the contents of the variable on its right to the object on its left.

3. Return type of main():-

In C++, main returns an integer type value to the operating system. So return type for main() is explicitly specified as int.Therefore every main( ) in c++ should end with a return 0 statement.

BASIC DATA TYPES IN C++



Built-in-type

1) Integral type : – The data types in this type are int and char. The modifiers signed, unsigned, long & short may be applied to character & integer basic data type. The size of int is 2 bytes and char is 1 byte.

2) void – void is used :

i) To specify the return type of a function when it is not returning any value.

ii) To indicate an empty argument list to a function.

Ex- void function1(void)

iii) In the declaration of generic pointers.

Ex- void *gp

A generic pointer can be assigned a pointer value of any basic data type.

Ex . int *ip // this is int pointer

gp = ip //assign int pointer to void.

A void pointer cannot be directly assigned to their type pointers in c++ we need to use cast operator.

Ex - void *ptr1;

char *ptr2;

ptr2 = ptr1; is allowed in c but not in c++.

ptr2 = (char *)ptr1; is the correct statement.

3) **Floating type:**

The data types in this are float & double. The size of the float is 4 byte and double is 8 byte. The modifier long can be applied to double & the size of long double is 10 byte.

User-defined type:

i) The user-defined data type **structure** and **union** are same as that of C.

ii) **Classes** – Class is a user defined data type which can be used just like any other basic data type once declared. The class variables are known as objects.

iii) **Enumeration**

a) An enumerated data type is another user defined type which provides a way of attaching names to numbers to increase simplicity of the code.

b) It uses enum keyword which automatically enumerates a list of words by assigning them values 0, 1, 2,…..etc.
c) **Syntax**:-

```
enum shape  {
                circle, square, triangle;
                }
```

Now shape becomes a new type name & we can declare new variables of this type.
Ex . shape oval;
d) In C++, enumerated data type has its own separate type. Therefore c++ does not permit an int value to be automatically converted to an enum value.
Ex. shape shapes1 = triangle, // is allowed
shape shape1 = 2; // Error in c++
shape shape1 = (shape)2; //ok
e) By default, enumerators are assigned integer values starting with 0, but we can over-ride the default value by assigning some other value.
Ex enum colour {red, blue, pink = 3}; //it will assign red to o, blue to 1, & pink to 3 or enum colour {red = 5, blue, green}; //it will assign red to 5, blue to 6 & green to 7.

Derived Data types:

1) Arrays

An array in c++ is similar to that in c, the only difference is the way character arrays are initialized. In c++, the size should be one larger than the number of character in the string where in c, it is exact same as the length of string constant.
Ex - char string1[3] = “ab”; // in c++
char string1[2] = “ab”; // in c.

2) Functions

Functions in c++ are different than in c there is lots of modification in functions in c++ due to object orientated concepts in c++.

3) Pointers

Pointers are declared & initialized as in c.
Ex int * ip; // int pointer
ip = &x; //address of x through indirection
c++ adds the concept of constant pointer & pointer to a constant pointer.
char const *p2 = .HELLO.; // constant pointer
Next you have a summary of the basic fundamental data types in C++, as well as the range of values that can be represented with each one:

Name Description Size* Range*
char - 1byte; signed: -128 t127, unsigned: 0 to 255
short int (short)
short Integer - 2bytes; signed: -32768 to 32767, unsigned: 0 to65535
int Integer. 4bytes signed: - 2147483648 to 2147483647, unsigned: 0 to 4294967295
long int or (long)
Long integer. 4bytes signed: -2147483648 to 2147483647, unsigned: 0 to 4294967295
Bool - Boolean value. It can take one of two values: true or false.
1byte true or false

Float Floating point
number.- 4bytes 3.4e +/- 38 (7 digits)
double Double precision
floating point
number.

8bytes 1.7e +/- 308 (15 digits)
long double, Long double, precision floating point number. 8bytes 1.7e +/- 308 (15 digits)

VARIABLES IN C++

Declaration of variables.

C requires all the variables to be defined at the beginning of a scope. But c++ allows the declaration of variable anywhere in the scope. That means a variable can be declared right at the place of its first use. It makes the program easier to understand. In order to use a variable in C++, we must first declare it specifying which data type we want it to be. The syntax to declare a new variable is to write the specifier of the desired data type (like int, bool, float...) followed by a valid variable identifier.

Ex.: int a;
float mynumber;

These are two valid declarations of variables. The first one declares a variable of type int with the identifier a. The second one declares a variable of type float with the identifier mynumber. Once declared, the variables a and mynumber can be used within the rest of their scope in the program. If you are going to declare more than one variable of the same type, you can declare all of them in a single statement by separating their identifiers with commas.

Ex.: int a, b, c;

This declares three variables (a, b and c), all of them of type int, and has exactly the same meaning as: int a; int b; int c;

The integer data types can be char, short, long. Integer data type can be either signed or unsigned depending on the range of numbers needed to be represented. Signed types can represent both positive and negative values, whereas unsigned types can only represent positive values (and zero). This can be specified by using either the specifier signed or the specifier unsigned before the type name.

Ex.: unsigned short int Number;
signed int Balance;

By default, if we do not specify either signed or unsigned most compiler settings will assume the type to be signed, therefore instead of the second declaration above we could have written:
int Balance;
with exactly the same meaning (with or without the keyword signed). An exception to this general rule is the char type, which exists by itself and is considered a different fundamental data type from signed char and unsigned char, thought to store characters.
You should use either signed or unsigned if you intend to store numerical values in a char-sized variable. short and long can be used alone as type specifiers. In this case, they refer to their respective integer fundamental types: short is equivalent to short int and long is equivalent to long int. The following two variable declarations are equivalent:
short Year; short int Year;

Scope of variables:

All the variables that we intend to use in a program must have been declared with its type specifier in an earlier point in the code, like we did in the previous code at the beginning of the body of the function main when we declared that a, b, and result were of type int. A variable can be either of global or local scope. A global variable is a variable declared in the main body of the source code, outside all functions, while a local variable is one declared within the body of a function or a block. Global variables can be referred from anywhere in the code, even inside functions, whenever it is after its declaration. The scope of local variables is limited to the block enclosed in braces ({ })

where they are declared. For example, if they are declared at the beginning of the body of a function (like in function main) their scope is between its declaration point and the end of that function. In the example above, this means that if another function existed in addition to main, the local variables declared in main could not be accessed from the other function and vice versa.

Dynamic initialization of variables.

In c++, a variable can be initialized at run time using expressions at the place of declaration. This is referred to as dynamic initialization.

Ex int m = 10;

Here variable m is declared and initialized at the same time.

Reference variables

i) A reference variable provides an alias for a previously defined variable.

ii) Example:-

If we make the variable sum a reference to the variable total, then sum & total can be used

interchangeably to represent that variable.

iii) Reference variable is created as:- data type &reference name = variable name;

Ex int sum = 200; int &total = sum;

Here total is the alternative name declared to represent the variable sum. Both variable efer to the same data object in memory.

iv) A reference variable must be initialized at the time of declaration.

v) C++ assigns additional meaning to the symbol & here & is not an address operation.

The notation int & means referenceto int.

A major application of reference variable is in passing arguments to functions.

Ex void fun (int &x) // uses reference

```
{
x = x + 10; // x increment, so x also incremented.
}
int main( )
{int n = 10;
fun(n); // function call
}
```

When the function call fun(n) is executed, it will assign x to n i.e. int &x = n;

Therefore x and n are aliases & when function increments x. n is also incremented. This type of function call is called call by reference.

Introduction to strings:

Variables that can store non-numerical values that are longer than one single character are known as strings. The C++ language library provides support for strings through the standard string class. This is not a fundamental type, but it behaves in a similar way as fundamental types do in its most basic usage. A first difference with fundamental data types is that in order to declare and use objects (variables) of this type we need to include an additional header file in our source code: <string> and have access to the std namespace (which we already had in all our previous programs thanks to the using namespace statement).The following initialization formats are valid for strings:

string mystring = “This is a string”;

string mystring (“This is a string”);

Strings can also perform all the other basic operations that fundamental data types can, like being declared without an initial value and being assigned values during execution.

OPERATORS IN C++

Assignment (=):

The assignment operator assigns a value to a variable.a=5;
This statement assigns the integer value 5 to the variable a. The part at the left of the assignment operator (=) is known as the *lvalue* (left value) and the right one as the *rvalue* (right value). The lvalue has to be a variable whereas the rvalue can be a constant, a variable, the result of an operation or any combination of these. The most important rule when assigning is the *right-to-left* rule: The assignment operation always takes place from right to left, and never the other way:
a= b;
This statement assigns to variable a (the lvalue) the value contained in variable b (the rvalue). The value that was stored until this moment in a is not considered at all in this operation, and in fact that value is lost.

Arithmetic operators (+, -, *, /, %)

The five arithmetical operations supported by the C++ language are:
+ addition
- subtraction
* multiplication
/ division
% modulo
Operations of addition, subtraction, multiplication and division literally correspond with their respective mathematical operators. The only one that you might not be so used to see may be *modulo*; whose operator is the percentage sign (%). Modulo is the operation that gives the remainder of a division of two values. For example, if we write:
a = 11 %3;
The variable a will contain the value 2, since 2 is the remainder from dividing 11 between 3.

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

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

expression is equivalent to

value += increase; value = value + increase;
a -= 5; a = a - 5;
a /= b; a = a / b;
price *= units + 1; price = price * (units + 1);

Increase and decrease (++, --)

Shortening even more some expressions, the increase operator (++) and the decrease operator (--) increase or reduce by one the value stored in a variable. They are equivalent to +=1 and to -=1, respectively. Thus:
c++; c+=1; c=c+1;
are all equivalent in its functionality: the three of them increase by one the value of c.
A characteristic of this operator is that it can be used both as a prefix and as a suffix. That means that it can be written either before the variable identifier (++a) or after it (a++). Although in simple expressions like a++ or ++a both have exactly the same meaning, in other expressions in which the result of the increase or decrease operation is evaluated as a value in an outer expression they may have an important difference in their meaning: In the case that the increase operator is used as a prefix (++a) the value is increased before the result of the expression is evaluated and therefore the increased value is considered in the outer expression; in case that it is used as a suffix (a++) the value stored in a is increased after being evaluated and therefore the value stored before the increase operation is evaluated in the outer expression.

Relational and equality operators (==, !=, >, <, >=, <=):

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

Operators Meaning
== Equal to
!= Not equal to
> Greater than
< Less than
>= Greater than or equal to
<= Less than or equal to

Here there are some examples:
(7 == 5) // evaluates to false.
(5 > 4) // evaluates to true.
(3 != 2) // evaluates to true.
(6 >= 6) // evaluates to true.
(5 < 5) // evaluates to false.

Of course, instead of using only numeric constants, we can use any valid expression, including variables. Suppose that a=2, b=3 and c=6,
(a == 5) // evaluates to false since a is not equal to 5.
(a*b >= c) // evaluates to true since (2*3 >= 6) is true.
(b+4 > a*c) // evaluates to false since (3+4 > 2*6) is false.
((b=2) == a) // evaluates to true.

Be careful! The operator = (one equal sign) is not the same as the operator == (two equal signs), the first one is an assignment operator (assigns the value at its right to the variable at its left) and the other one (==) is the equality operator that compares whether both expressions in the two sides of it are equal to each other.
Thus, in the last expression ((b=2) == a), we first assigned the value 2 to b and then we compared it to a, that also stores the value 2, so the result of the operation is true.

Logical operators (!, &&, ||):

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

For example:
!(5 == 5) // evaluates to false because the expression at its right (5 == 5) is true.
!(6 <= 4) // evaluates to true because (6 <= 4) would be false.
!true // evaluates to false
!false // evaluates to true.

The logical operators && and || are used when evaluating two expressions to obtain a single relational result. The operator && corresponds with Boolean logical operation AND. This operation results true if both its two operands are true, and false otherwise. The following panel shows the result of operator && evaluating the expression a && b:

&& OPERATOR
a b a && b
true true true
true false false
false true false
false false false

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

|| OPERATOR

a	b	a || b
true	true	true
true	false	true
false	true	true
false	false	false

For example:
((5 == 5) && (3 > 6)) // evaluates to false (true && false).
((5 == 5) || (3 > 6)) // evaluates to true (true || false).

Conditional operator (?)
The conditional operator evaluates an expression returning a value if that expression is true and a different one if the expression is evaluated as false. Its format is:
condition ? result1 : result2
If condition is true the expression will return result1, if it is not it will return result2.
7==5 ? 4 : 3 // returns 3, since 7 is not equal to 5.
7==5+2 ? 4 : 3 // returns 4, since 7 is equal to 5+2.
5>3 ? a : b // returns the value of a, since 5 is greater than 3.
a>b ? a : b // returns whichever is greater, a or b.

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

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

Some of the new operators in c++ are-
:: Scope resolution operators.
::* pointer to member decelerator.
・ * pointer to member operator.
.* pointer to member operator.
delete memory release operator.
endl Line feed operator
new Memory allocation operator.
stew Field width operator.

Scope resolution Operator.
1) C++ is a block - structured language. The scope of the variable extends from the point of its declaration till the end of the block containing the declaration.
2) Consider following program.
...
....
{ int x = 1;

===
}
=====
{ int x = 2;
} =====
The two declaration of x refer to two different memory locations containing different values. Blocks in c++ are often nested. Declaration in a inner block hides a declaration of the same
variable in an outer block.
3) In C, the global version of a variable cannot be accessed from within the inner block. C++ resolves this problem by using scope resolution operator (::), because this operator allows access to the global version of a variable.
4) Consider following program.
# include<iostream.h >
int m = 10;
int main ( )
{int m = 20;
{int k = m;
int m = 30;
cout << .K = . <<k;
cout << .m = .<<m;
cout << : : m = . << : : m;
}
cout << .m. = .<< m;
cout << .::m = <<::m;
return 0;
}
The output of program is
k = 20
m = 30
:: m = 10
m = 20
:: m =10
In the above program m is declared at three places. And ::m will always refer to global m.
5) A major application of the scope resolution operator is in the classes to identify the class to which a member functions belongs.

Member dereferencing operators .

C++ permits us to define a class containing various types of data & functions as members. C++ also permits us to access the class members through pointers. C++ provides a set of three pointer. C++ provides a set of three pointers to member operators.
1) ::* - To access a pointer to a member of a class.
2) .* - To access a member using object name & a pointer to that member.
3) ・ * - To access a member using a pointer in the object & a pointer to the member.

Memory management operators.

1) We use dynamic allocation techniques when it is not known in advance how much of memory space is needed. C++ supports two unary operators **new** and **delete** that perform the task of allocating & freeing the memory.
2) An object can be created by using new and destroyed by using delete. A data object created inside a block with new, will remain existence until it is explicitly destroyed by using delete.

3) It takes following form.
variable = new data type
The new operator allocated sufficient memory to hold a data object of type data-type & returns the address of the object.
EX p = new int.
Where p is a pointer of type int. Here p must have already been declared as pointer of appropriate types.
4) New can be used to create a memory space for any data type including user defined type such all array, classes etc.
Ex int * p = new int [10]; Creates a memory space for an array of 10 integers.
5) When a data object is no longer needed, it is destroyed to release the memory space for reuse. The general form is delete variable. If we want to free a dynamically allocated array, we must use following form.
delete [size] variable; The size specifies the no of elements in the array to be freed.
6) The new operator has following advantages over the function
malloc() in c -.
i) It automatically computes the size of the data object. No need to use sizeOf()
ii) If automatically returns the correct pointer type, so that there is no need to use a type cast.
iii) new and delete operators can be overloaded.
iv) It is possible to initialize the object while creating the memory space.

Manipulators:

Manipulators are operators that are used to format the data display. There are two important manipulators.
1) endl
2) stew
1) **endl** : -
This manipulator is used to insert a linefeed into an output. It has same effect as using .\n. for newline.
Ex cout<< .a. << a << endl <<.n=. <<n;
<< endl<<.p=.<<p<<endl;
The output is
a = 2568
n = 34
p = 275
2) **Setw** : -
With the stew, we can specify a common field width for all the numbers and force them to print with right alignment.
EX cout<<stew (5) <<sum<<endl;
The manipulator setw(5) specifies a field width of 5 for printing the value of variable sum the value is right justified.
3 5 6

Type cast operator.

C++ permits explicit type conversion of variables or expressions using the type cast operator.
Syntax . type name (expression)
Ex avg = sum/float(i)
Here a type name behaves as if it is a function for converting values to a designated type.

CONTROL STATEMENTS

Structure

Introduction :
if – else Statement
Nested ifs :
The switch Statement
The for Loop
The while Loop
The do-while Loop
Using break to Exit a Loop
Using the goto Statement

INTRODUCTION :

This module discusses the statements that control a program's flow of execution. There are three categories of selection statements, which include the if and the switch; iteration statements, which include the for, while, and do-while loops; and jump statements, which include break, continue, return, and goto. Except for return, which is discussed later in this book, the remaining control statements, including the if and for statements to which you have already had a brief introduction, are examined here.

IF – ELSE STATEMENT

The complete form of the if statement is

```
if(expression) statement
else statement
```

where the targets of the if and else are single statements. The else clause is optional. The targets of both the if and else can also be blocks of statements. The general form of the if using blocks of statements is

```
if(expression)
{
statement sequence
}
else
{
statement sequence
}
```

If the conditional expression is true, the target of the if will be executed; otherwise, the target of the else, if it exists, will be executed. At no time will both be executed. The conditional expression controlling the if may be any type of valid C++ expression that produces a true or false result. This program uses the 'if' statement to determine whether
the user's guess matches the magic number. If it does, the message is printed on the screen. Taking the Magic Number program further, the next version uses the else to print a message when the wrong number is picked:

```
#include<iostream>
#include<cstdlib>
int main()
{
int magic;// magic number
int guess;// users guess
magic = rand(); // get a random number
cout << "Enter your guess"
cin >> guess;
if (guess == magic) cout << "Right";
else cout<< " sorry , your are wrong";
return 0;
}
```

NESTED IFS :

A nested if is an if statement that is the target of another if or else. Nested ifs are very common in programming. The main thing to remember about nested ifs in C++ is that an else statement always refers to the nearest if statement that is within the same block as the else and not already associated with an else. The general form of the nested if using blocks of statements is

```
if(expression)
{
statement sequence
if(expression)
{
statement sequence
}
else
{
statement sequence
}
}
else
{
statement sequence
}
```

Here is an example

```
if(i)
{
if(j) result =1;
if(k) result =2;
else result = 3;
}
else result = 4;
```

The final else is not associated with if(j) (even though it is the closest if without an else), because it is not in the same block. Rather, the final else is associated with if(i). The inner else is associated with if(k) because that is the nearest if. You can use a nested if to add a further improvement to the Magic Number program. This addition provides the player with feedback about a wrong guess.

THE SWITCH STATEMENT

The second of C++'s selection statements is the switch. The switch provides for a multiway branch. Thus, it enables a program to select among several alternatives. Although a series of nested if statements can perform multiway tests, for many situations the switch is a more efficient approach. It works like this: the value of an expression is successively tested against a list of constants. When a match is found, the statement sequence associated with
that match is executed. The general form of the switch statement is

```
Switch(expression)
{ case constant 1 :
Statement sequence
Break;
case constant 2 :
Statement sequence
Break;
case constant 3 :
Statement sequence
```

Break;
...
Default:
Statement sequence
}
The switch expression must evaluate to either a character or an integer value. (Floatingpoint expressions, for example, are not allowed.) Frequently, the expression controlling the switch is simply a variable. The case constants must be integer or character literals.
The default statement sequence is performed if no matches are found. The default is optional; if it is not present, no action takes place if all matches fail. When a match is found, the statements associated with that case are executed until the break is encountered or, in a concluding case or default statement, until the end of the switch is reached. There are four important things to know about the switch statement:
The switch differs from the if in that switch can test only for equality (that is, for matches between the switch expression and the case constants), whereas the if conditional expression can be of any type.
No two case constants in the same switch can have identical values. Of course, a switch statement enclosed by an outer switch may have case constants that are the same. A switch statement is usually more efficient than nested ifs. The statement sequences associated with each case are not blocks. However, the entire switch statement does define a block.
The following program demonstrates the switch. It asks for a number between 1 and 3, inclusive. It then displays a proverb linked to that number. Any other number causes an error message to be displayed.

```
#include <iostream>
using namespace std;
int main()
{
int num;
cout<< “Enter a number from 1 to 3”;
cin >>num;
switch(num)
{
case 1:
cout << “ A rolling stone gathers no moss.\n”;
break;
case 2:
cout<< “ A bird in hand is worth two in the bush.\n”;
break;
case 3:
cout<< “ A full and his money soon parted.\n”;
break;
default :
cout<< “ you must enter 1 , 2 or 3.\n”;
break;
}
return 0;
}
```

Here is a sample run
Enter a number from 1 to 32
A bird in hand is worth two in the bush.

THE FOR LOOP

You have been using a simple form of the for loop since Module 1. You might be surprised at just how powerful and flexible the for loop is. Let's begin by reviewing the basics, starting with the most traditional forms of the for. The general form of the for loop for repeating a single statement is

for(initialization; expression; increment) statement;

For repeating a block, the general form is

```
for(initialization; expression; increment)
{
statement sequence
}
```

The initialization is usually an assignment statement that sets the initial value of the loop control variable, which acts as the counter that controls the loop. The expression is a conditional expression that determines whether the loop will repeat. The increment defines the amount by which the loop control variable will change each time the loop is repeated. Notice that these three major sections of the loop must be separated by semicolons. The for loop will continue to execute as long as the conditional expression tests true. Once the condition becomes false, the loop will exit, and program execution will resume on the statement following the for block. The following program uses a for loop to print the square roots of the numbers between 1 and 99. Notice that in this example, the loop control variable is called num.

```
#include <iostream>
#inclume<cmath>
int main()
{
int num;
double sq_root;
for(num=1;num<100;num++)
{
sq_root=sqrt((double)num);
cout<<num<<""<<sq_root<<'\n';
}
return 0;
}
```

This program uses the standard function sqrt( ). As explained in Module 2, the sqrt( ) function returns the square root of its argument. The argument must be of type double, and the function returns a value of type double. The header <cmath> is required.

THE WHILE LOOP

Another loop is the while. The general form of the while loop is while(expression) statement; where statement may be a single statement or a block of statements. The expression defines the condition that controls the loop, and it can be any valid expression. The statement is performed while the condition is true. When the condition becomes false, program control passes to the line immediately following the loop. The following program illustrates this characteristic of the while loop. It displays a line of periods. The number of periods displayed is equal to the value entered by the user. The program does not allow lines longer than 80 characters. The test for a permissible number of periods is performed inside the loop's conditional expression, not outside of it.

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

```
cout<<"Enter length(1 to 79)";
cin>>len;
while(len>0 $$ len<80)
{
cout<<'.';
len--;
}
return 0;
}
```

If len is out of range, then the while loop will not execute even once. Otherwise, the loop executes until len reaches zero. There need not be any statements at all in the body of the while loop. Here is an example: while(rand() != 100) ; This loop iterates until the random number generated by rand( ) equals 100.

THE DO-WHILE LOOP

The last of C++'s loops is the do-while. Unlike the for and the while loops, in which the condition is tested at the top of the loop, the do-while loop checks its condition at the bottom of the loop. This means that a do-while loop will always execute at least once. The general form of the do-while loop is do { statements; } while(condition); Although the braces are not necessary when only one statement is present, they are often used to improve readability of the do-while construct, thus preventing confusion with the while. The do-while loop executes as long as the conditional expression is true. The following program loops until the number 100 is entered:

```
#include<iostream>
using namespace std;
int main()
{
int num;
do
{
cout << "Enter a number (100 to stop):";
cin>> num;
}
while(num!=100);
return 0;
}
```

USING BREAK TO EXIT A LOOP

It is possible to force an immediate exit from a loop, bypassing the loop's conditional test, by using the break statement. When the break statement is encountered inside a loop, the loop is immediately terminated, and program control resumes at the next statement following the loop. Here is a simple example:

```
#include<iostream>
using namespace std;
int main()
{
int t;
for(t=0;t<100;t++)
{
if(t==10) break;
cout<<t<< ' ';
}
return 0;
```

```
}
```

The output from the program is shown here: 0 1 2 3 4 5 6 7 8 9. As the output illustrates, this program prints only the numbers 0 through 9 on the screen before ending. It will not go to 100, because the break statement will cause it to terminate early.

Using continue

It is possible to force an early iteration of a loop, bypassing the loop's normal control structure. This is accomplished using continue. The continue statement forces the next iteration of the loop to take place, skipping any code between itself and the conditional expression that controls the loop. For example, the following program prints the even numbers between 0 and 100:

```
#include<iostream>
using namespace std;
int main()
{
int x;
for(x=0;x<100;x++)
{
if(x%2) continue;
cout<<x<< ' ';
}
return 0;
}
```

Only even numbers are printed, because an odd number will cause the continue statement to execute, resulting in early iteration of the loop, bypassing the cout statement. Remember that the % operator produces the remainder of an integer division. Thus, when x is odd, the remainder is 1, which is true. When x is even, the remainder is 0, which is false. In while and do-while loops, a continue statement will cause control to go directly to the conditional expression and then continue the looping process. In the case of the for, the increment part of the loop is performed, then the conditional expression is executed, and then the loop continues.

USING THE GOTO STATEMENT

The goto is C++'s unconditional jump statement. Thus, when encountered, program flow jumps to the location specified by the goto.

There are times when the use of the goto can clarify program flow rather than confuse it. The goto requires a label for operation. A label is a valid C++ identifier followed by a colon. Furthermore, the label must be in the same function as the goto that uses it. For example, a loop from 1 to 100 could be written using a goto and a label, as shown here:

```
x=1;
loop1:
x++;
if(x<100) goto loop1;
```

One good use for the goto is to exit from a deeply nested routine.

For example, consider the following code fragment:

```
for(…){
for(…){
while(…){
if(…)goto stop
.
.
.
}
}
```

```
}
stop:
cout<<"Enter in program.\n";
```

Eliminating the goto would force a number of additional tests to be performed. A simple break statement would not work here, because it would only cause the program to exit from the innermost loop.

LECTURE-11

Object Oriented Programming Using C++

Why OOP language?

Advantages of OOP languages are:

(i) OOP introduces the concept of data hiding & data encapsulation, because of which user is exposed to minimal data, thus creating safer programs.

(ii) OOP introduces the concept of data abstraction because of which the user can only see the required features and does not need to go background details.

(iii) OOP also introduces the concept of code reusability and inheritance,which makes the software development faster.

(iv) OOP concept of polymorphism can allow a function with same name work differently for different classes. So user can use same functions name for different purposes.

Class Vs Structure

Class is a collective representation of a set of member variable and member function under one name.

Structure is a collection of member variables and function of various data types.

Difference between Class and Structure

Class	Structure
• By default the member of the classes are private.	• By default the members of Structures are public.

Friend Function:

It is a non-member function that can access the private members of the class to which it is declared as friend.

According to OOP concept of data hiding or data encapsulation, no outside function can access the private data in a class. Since friend function is a clear violation of this concept. It is rarely used, so as to reduce data security ricks.

LECTURE-12

Basic Object Oriented concept

What is class?

Class is a collection of similar types of objects, where each class having similar properties and types.

Ex: Bird, Animal

Need of class and object

Class supports data encapsulation and data abstraction which leads C++ a more secure programming language. Apart from these features, the class also has private and public section, where we put the very important data in private section and comparatively less important data in public section. These two are known as access specifiers.

Features of OOPS

The main features of OOP are class, object, data hiding and encapsulation, inheritance, polymorphism and message passing.

Class and Object

Class is a collection of similar types of objects. For example if college, school, institutions are three objects then, combinly the three objects form a class and the class name is Education.

Ex2:

If elephants, tigers, lions and cats will be considered as objects then the class for these object will be Animal.

You can consider the class and object name as per your wish and always keep in mind that these objects are formed from the class. So class is imagined as a whole and objects are part of it.

Data hiding

Putting the data in private section can only be accessed by using object and the function associated with it. This feature of OOP is known as data hiding.

Encapsulation

Wrapping around data and functions inside a class is known as encapsulation. This feature makes program easy to understand.

Inheritance

The code reusabilty features for programming which is also known as inheritance. So inheritance is the property by which one class can get the properties of another class.

Polymorphism

Poly means many and morphism means form. It's a greek word whose meaning is one name multiple form. This feature allows to write same name for functions and operators but the meaning for each one varies in different places. Its of two type i.e. run type polymorpism and compile time polymorphism.

Dynamic binding

Asociating function call during program execuation is known as dynamic binding.

Message passing

Objects communicates with each other through message is known as message passing. Data is transferred from one object to another through object and functions.

Note: Most frequently variables are put in private sections because these are very sensitive data for the real world purpose. Generally functions are put in public section which are mainly used for accessing these private data.

LECTURE-13

Public construction

Member functions:

All the the declared functions and varibles are part of the class and hence members of the corresponding class. Those are declared and defined outside the class are known as not the members of the class. So all the functions declared inside the class are known as member functions.

Ex:

```
#include<iostream.h>

Using namespace std;

Class Abc

{

Void show()

{

Cout<<"This is inside a member function";
```

```
}
};
Void display()
{
Cout<<"We are no more inside a member function";
}
void main()
{
Abc obj;
Obj.show();
}
```

Member function definition and declaration

When member function is declared it is not necessary to complete its defination inside the class, for that purpose the definition is left undefined for full definition outside the class. The main purpose is to main the C++ program more redable and understandable. When the definition itself is some thousands line of code. Then the code itself becomes clumsy for understanding and for that purpose we write the function outside the class. Referring to the previous example

The Function declaration ends with semicolon(;) where definition don't have semicolon

function declaration

Syntax:

Returntype functionname(parameter list);

Function Defination

Syntax:

Returntype classname::functionname(parameter list**)**

{

}

Function Calling

Main function is a separate function other than the member function, so it does not have any permission to call the member function directly. Thus calling a member function need the help of object and function is called with the help of the corresponding object for which the function is a member.

Syntax function calling:

Objectname.functionname(actual parameters);

Ex:

```
#include<iostream.h>
Using namespace std;
Class Abc
{
Public:
Void show(); // Declaration part
};
Void Abc::show()      // Defination part
{
Cout<<"This is inside a member function";
}
void main()
{
Abc obj;
Obj.show();
}
```

Need of defining function ouside the Class

For better readable purpose generally functions are defined outside the class.

Inline Function

Inline functions are generally works like macro, where the function call replaces the function with its definations. This work is done during compilation not during run time execuation. Inline function reduces the accessing overhead of member function.

Syntax:

Inline return type classname::function name(arguments)

Need of Inline function

Faster in program execution

Rules for Inline Function

1. Use when function contains very few lines of code.
2. Inline function should not be used where loop exist.

How to define a member function inside or outside a class

Member function can only be accessed from outside class using class name or object name. When object is used for accessing purpose then dot(.) operaror is used. Similarly when class name is used then scope resolution operator(::) is used for accessing any function.

The number and type of parameter remains the same as during declaration of the function. But the parameter used in declaration of the function becomes dummy or formal parameters.

The parameters that are passed to a function during its declaration is known as actual parameter.

LECTURE-14

Static members
Class members which are declared as static are known as Static members.
Always Static variable shares same memory location.
Syntax:
Static int a;
Static void show();

Static Variables
The data members which are declared static are known as static variables.

Program

```
#include<iostream>
using namespace std;
class num
{
static int c;
public:
void count()
{
++c;
cout<<"value of c is"<<c;
}
};
int num::c=10;
int main()
{
num obj1,obj2;
obj1.count();
obj2.count();
}
```

Output:

11

12

Note:

1. Here static data member variables must be initialised otherwise the linker will generate an error.
2. The memory for static data members is allocated only once

3. Only one copy of static member variable is created for the whole class for any number of objects. All the objects have common static data members.

Static Member Function

The member function which are declared static are known as static member function. Only static keyword before a member function makes a member function static.
Static member function can only access static members of the corresponding class.

Note:
Private static function must be invoked using static public function

LECTURE-15
Inheritance

One of the best feature that OOP provides is inheritance, where a new class can be constructed from an existing class. The class which is newly constructed is known as derived class and the class from which new class is constructed is known as Base class.

Derived classes

We can derive as many new classes from an existing class and there is no limitation of it.

Inheritance

Inheritance is the property of OOP language which provides code reusability and hence reduces lines of code in the program.

Types of Inheritance

Single Inheritance:

When a single class is derived from on single base class is known as single inheritance.

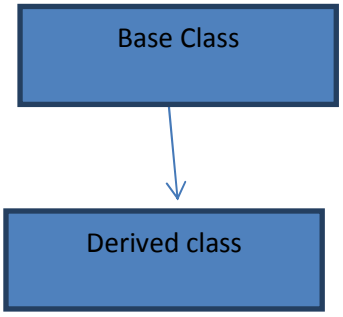


Fig. Single Inheritance

Multiple Inheritance:

When one single class is derived from more than one base class is known as multiple inheritance.

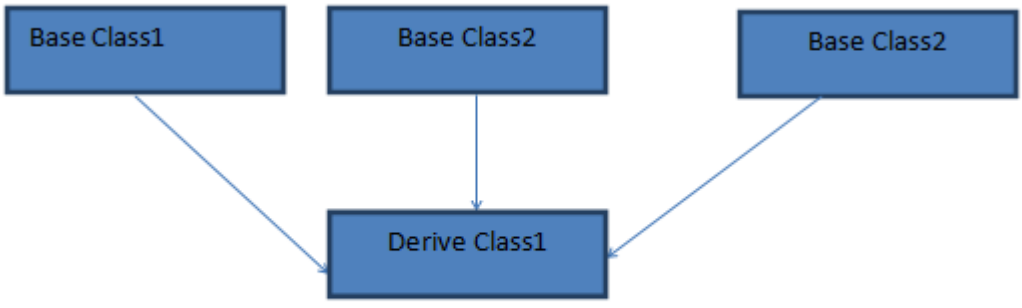


Fig. Multiple Inheritance

Multilevel Inheritance:

When a class is derived from a class which is derived from some other class for which this class is a derived class is known as multilevel Inheritance.

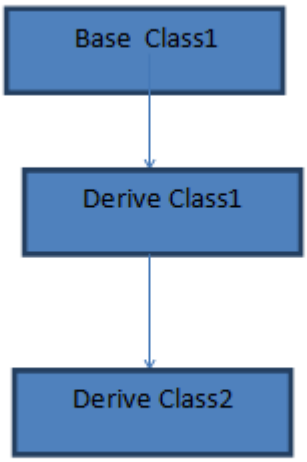


Fig: Multilevel Inheritance

Hierarchical Inheritance:

When more than one class is derived from an existing class is known as Hierarchical inheritance.

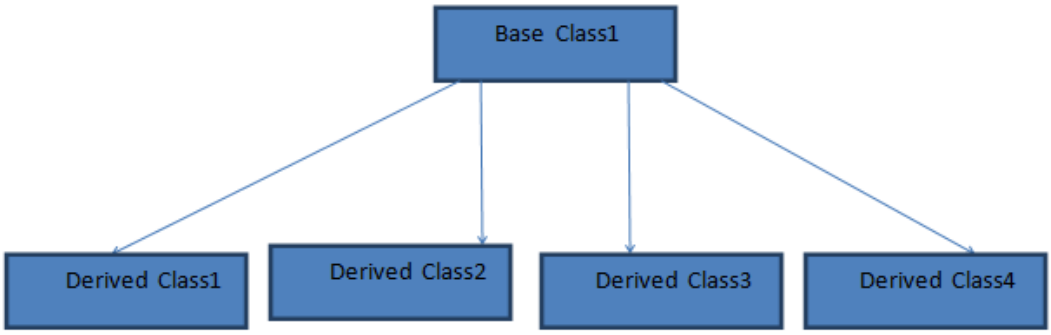


Fig: Hierarchical Inheritance

Hybrid Inheritance:

An inheritance which is mix of more than one type of inheritance is known as Hybrid Inheritance.

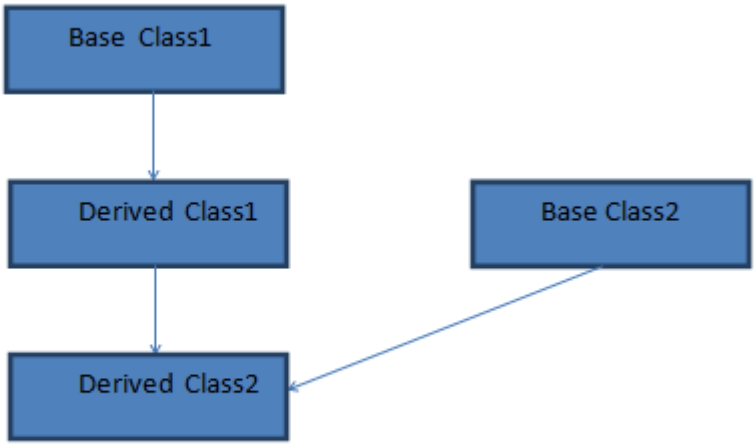


Fig: Hybrid Inheritance

Multipath Inheritance:

When multiple path exists for class members for multiple or hybrid inheritance for which duplicity arises for member functions and data members is known as multipath inheritance.

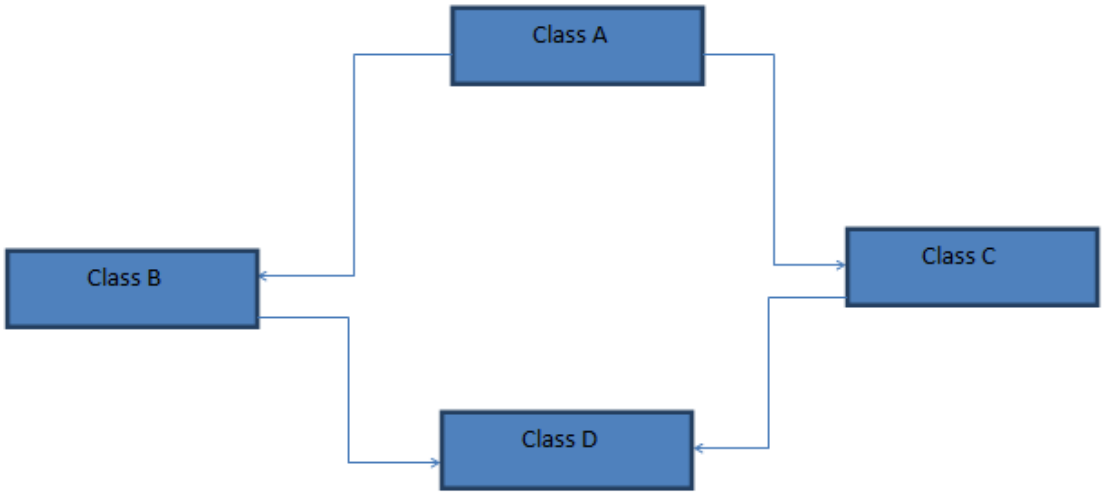


Fig: Multipath Inheritance

Inheriting in different access Modes

A class can be inheritated in three different modes i.e. Public, Private and Protected mode.

Note: Only Public and Protected members can be inheritated to derived class. In any case private members can't be inherited

Private inheritance

When a class is derived in private mode then all the public and protected members will be private for the derived class.

Public inheritance

When a class is derived in public mode then the protected members will be protected and public members will be public for the derived class.

Protected inheritance

When a class is derived in protected mode then all the protected members as well as the public members will be protected for the derived class.

The relation between mode of inheritance and its parent class scope can be classified as follows

Inhertance/Scope	Private	Protected	Public
private	Not inheritable	private	Private
protected	Not inheritable	protected	Protected
public	Not inheritable	protected	Public

Table: Access specifiers

LECTURE-16

Multipath Inheritance

This is one type of inheritance where a class can be derived from more than one base class and that base class is a derived class for some more parent class. In this chain of class hierarchy, a base class is inherited more than once which result in duplication of members present in the subsequent classes.

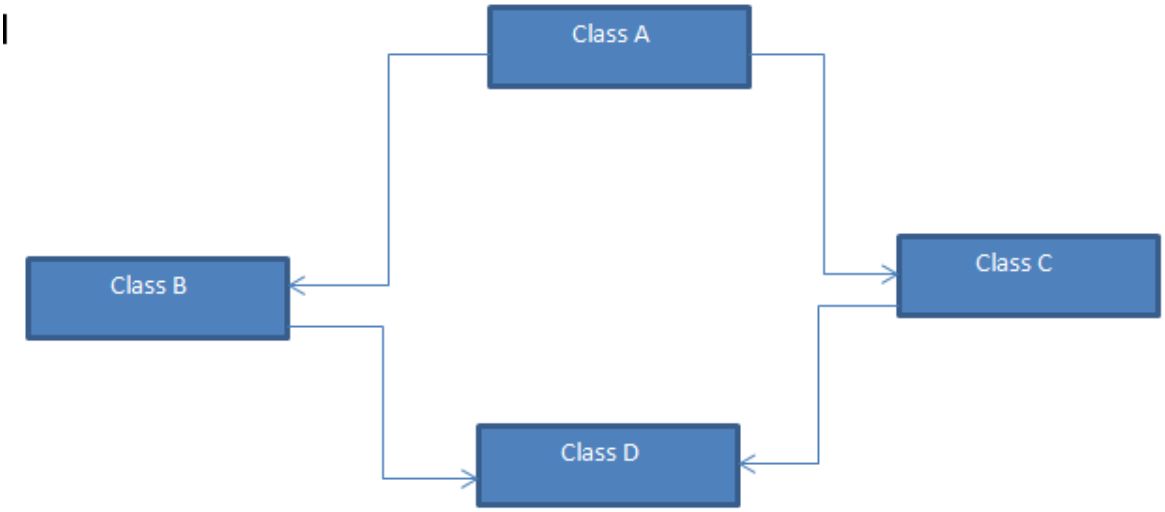


Fig. Multipath Inheritance

Here in this case, class A is derived twice by class B and class C, which result in duplication of inheritaed data members in the derived class i.e. class D.

This issue can be tackled by Virtual Base class.

Virtual Base Class

The ambiguity which aries in multipath inheritance is overcome by virtual base class.

Virtual class declaration

Syntax:

```
Class A{
//class body
}
Class B: virtual public A
```

```
{

}
```

Note1: When classes has been declared virtual in inheritance then the compiler will take necessary precaution to avoid the duplication of members. Generally we make a class virtual when it is used more than once in the program.

Note2: virtual keyword can appear after public but it shoud not come after the class name.

Structure and Union work similar to class, the following example will illustrate the concept of struct, class and union.

Inheritance using structure

```
#include<iostream>

using namespace std;

struct Base

{

int A;

};


struct Derived:public Base

{

int B;

   //Derived()

   //{B=90;}

void display();

};

void Derived::display()

{

cout<<endl<<"A = "<<A<<endl;

cout<<endl<<"B ="<<B<<endl;
```

```
}
int main()
{
   Derived D;
   D.A = 111;
D.display();
return 0;
}
```

LECTURE-17

Polymorphism

Poly means many and morphism means form and this is a greek word.
In OOP function overloading and operating are example of polymorphism. Polymorphism means one name multiple form.
Note: Virtual function is also another example of polymorphism which we will teach you later.
Polymorphism is of two types i.e. static polymorphism or early binding and dynamic polymorphism or run time polymorphism.

Static Polymorphism

Function is associated with object during program compilation is known as static binding or static binding.

Ex: operator overloading and function overloading

Dynamic Polymorphism

Function is associated with object during program execution is known as late binding or dynamic binding.

Ex: Virtual Function

LECTURE-18

Object slicing

In inheritance when one object for derived class is declared and the derived class object is assigned to base class object, then the value of the derived class object will be assigned to base class data member omitting all others data members value in deried class object.

Base initialization

Base class data members can be initialised in the derived class following the scope of the derived members.If the base class members are derived to public part of the derived class then we can initialize it directly otherwise it needs the help of member function of base class or derived class for the same.

Note: Private mebers of the base class can not be inherited to the derived class, so can’t be initialised using derive class constructor or any member function.

LECTURE - 19

Virtual functions

It is one type of run type polymorphism where the function will be linked to object at run time.
For making a function virtual we need to preceed the function declaration with keyword virtual.

Rules for Virtual Function

1. Virtual functions should not be static and must be member of a class.
2. A virtual function may be declared as friend for other class. Object point can acces virtual functions.
3. Constructor cannot be declared as virtual, but destructor can be declared as virtual.
4. Virtula function must be defined in public section of the class.It is also possible to declare virtual function outside the class. Insuch a case, the declaration is done inside the class and definition is done outside the class. The virtual keyword is used in the declaration and not in the function defination.
5. It is also possible to return a value from virtual function like other function.

Pure Virtual Function

In certain situations base class member functions are rarely used for any operation, such member function are known as do-nothing functions or dummy functions or pure virtual functions.After declarion of pure virtual functions the class becomes abstract class. It can't be used for declaring any object. Any attempt to declare object will show some error.

Syntax:
Virtual void display()=0; // pure virtual function

Abstract Class

Abstarct classes are used to create new classes from it. It works like skeleton class for other derived classes. It can be used as a base class only. Existance of pure virtual function inside a class makes it an abstract class.

Pointers and Class

Pointer is a variable which holds address of another variable. Along with all object oriented programming concept in C++ also uses pointer for efficient and faster programming.

Void pointer

A pointer of type void is known as void pointer. It can point to address of any type of variable after proper type casting. It is possible to declare void pointer where void data type is not possible.

Programs1 :

```
#include<iostream>
using namespace std;
//~ class one{
        //~ public:
        //~ int p;
        //~ void *x;
        //~ *(int *) x=12;
        //~ void display()
        //~ {cout<<"value of "<<x<<"is";}
        //~
        //~ };
//~ class two{};
int p;
void *pt=&p;
class book{
        public:
char name[25];
char author[25];
int pages;
};
class book *ptr;

int main(){
        //~ one obj;
        //~ obj.display();
```

```
        *(int *)pt=12;
        int *wld;
        //~ *pt=12; //void*' is not a pointer-to-object type
        cout<<"p="<<p;
        for(int k=0;k<10;k++)
        cout<<"wld"<<wld[k]<<"\n";

        cout<<"ptr"<<ptr->name;

}
```

Wild pointer

When a pointer looses its identity when the variable whose address it points to is deleted without freeing the pointer pointing address the corresponding pointer is known as Wild pointer.

By definition a wild pointer points to an unallocated memory location.

A pointer becomes wild because of the following reasons.

1. No initialisation of pointer
2. Accessing destroyed data
3. Pointer alteration

Program 1

```
#include<iostream>

using namespace std;

int main()

{

        int *x;

        for(int k=0;k<10;k++)

        cout<<x[k]<<"\n";

        return 0;

}
```

Program 2

```
#include<iostream>
using namespace std;
char *instring()
{
	char str[]="cpp";
	return str;
}
char *inchar()
{
	char g='D';
	g='D';
	return&g;
}
int main()
{
	char *ps,*pc;
	ps=instring();
	pc=inchar();
	cout<<"String"<<*ps<<endl;
	cout<<"Character"<<*pc<<endl;
	return 0;
}
```

LECTURE – 20

Function Prototyping

Function prototype provides an interface to the compiler by giving following details, namely argument-list and the type of return values.

Syntax: type function-name(argument-list);
Example: float area(float length, float width);
However, float area(float length, width) is illegal.

Function declaration provides the names of argument-list as dummy variables.

Example: float area(float, float);

Function definition provides the names of argument-list as variables because the names are used inside the function.

Example: float area(float l, float w)
{
float a = l × w;
…..
…..
}

To invoke the function area() in a program, we write
float rectangle = area(len, wid);
where len and wid are known as the actual parameters.

NOTE: It is possible that the argument-list is empty.

Example: float show();
This is same as float show(void);

Linking

To create a file, we write
Syntax:vi file-name
Example: vi vssut.c

To compile a file, we write

Syntax: cc file-name
Example: cc vssut.c

Then, the compiler produces an object file vssut.o and link with the library functions to make an executable file. The default executable file is a.out.

A program contains multiple files can be compiled at a time.
Syntax: cc file-name1 object-file-name
Example: cc cse.c it.o

The above statement compiles the file cse.c and links it with the it.o that is previously compiled.

LECTURE – 21

Operator Overloading

One of the exciting features of C++ is to permit the user to carry out different operations (like addition) using user-defined types with the same syntax as applied to the basic types.

The ability to provide the operators with some special meaning is known as operator overloading.

However, we cannot overload the following operators, namely size operator (sizeof), conditional operator (?:), scope resolution operator (::), object type operator (typeid) and member access operators (. (dot operator), .* (pointer to member operator)).

Operator function is to specify the relation with class in which the operator is applied. It can be a non-static member functions or friend functions.

Syntax: return-type class-name :: operator opr (argument-list)

```
{
  …..
  …..
}
```

Where return-type is the type of value returned
class-name is the name of the class
opr is the operator that is overloaded

If the operator function is a friend function, then it has one argument for unary operators and two for binary operators.

In contrary, if the operator function is a member function, then it has no arguments for unary operators and one for binary operators.

The process of overloading is as follows:

1. Create a class that defines the data type used in the overloading operation.
2. The operator function is declared in the public section of the class.
3. The operator function is defined to carry out the required operation.

Overloading Unary Operators

Unary operator takes one operand. For example unary minus (-) takes one operand. This operator changes the sign of an operand that is plus (+) to minus and vice-versa.

```
#include<iostream.h>
classvssut
{
        intcse, it, mca;
        public:
                void input(int c, int i, int m);
                void output(void);
                void operator-();
};
voidvssut :: input(int c, int i, int m)
{
        cse = c; it = i; mca = m;
}
voidvssut :: output(void)
{
        cout<<cse<< “\t” << it << “\t” <<mca<< “\n”;
}
voidvssut :: operator-()
{
        cse = -cse; it = -it; mca = -mca;
}
int main()
{
        vssut v;
        v.input(-40, 20, -15);
        cout<< “Output: ”;
        v.output();
        -v;
        cout<< “Output: ”;
        v.output();
        return 0;
}
```

OUTPUT

```
Output: -40 20 -15
Output: 40 -20 15
```

```
#include<iostream.h>
classvssut
{
        intcse, it;
        public:
                vssut(){}
                vssut(float r, float i);
                {cse = r; it = i;}
                vssut operator+(vssut);
                void output(void);
};
vssutvssut :: operator+(vssut v)
{
        vssut temp;
        temp.cse = cse + v.cse;
        temp.it = it + v.it;
        return temp;
}
voidvssut :: output(void)
{
        cout<<cse<< "+ j" << it << "\n";
}
int main()
{
        vssut v1,v2,v3;
        v1 = vssut(4.5, 8.5);
        v2 = vssut(7.2, 12.3);
        v3 = v1 + v2;
        cout<< "v1 =";
        v1.output();
        cout<< "v2 =";
        v2.output();
        cout<< "v3 =";
        v3.output();
        return 0;
}
```

OUTPUT

```
v1 = 4.5 + j8.5
v2 = 7.2 + j12.3
v3 = 11.7 + j20.8
```

LECTURE- 22

Rules of Overloading Operators

1. The overloaded operator must have at least one operand.
2. Some operators cannot be overloaded.
3. Only existing operators can be overloaded. New operators cannot be created.
4. The meaning of an operator cannot be changed.
5. Overloaded operators cannot be overridden.
6. The friend function cannot be overloaded for few operators (like =, (), [], ->)
7. Unary operators take no argument when overloaded by a member function. But it takes one argument when overloaded by a friend function.
8. Binary operators take one argument when overloaded by a member function. But it takes two arguments when overloaded by a friend function.

Misuse of Operator Overloading

```
#include<iostream.h>
classvssut
{
        intcse;
        public:
                vssut() {cse = 0;}
                vssut(int it) {cse = it;}
                vssut operator+(vssutmca)
                {
                        vssutdept;
                        dept.cse = cse – mca.cse;
                        returndept;
                }
                void output()
                { cout<< "\n cse = " <<cse;}
};
int main()
{
        vssut v1(20), v2(15), v3;
        v3 = v1 + v2;
        v3.output();
        return 0;
}
```

OUTPUT

```
cse = 5
```

Overloading with Friend Function

Friend function gives better flexibility than the member function of the class. The main difference between member function and friend function is that the member function takes arguments explicitly. However, friend function requires the arguments to be explicitely passed.

Syntax: friend return-type operator opr (argument-list)
{
…..

…..
}

```
#include<iostream.h>
classvssut
{
        floatcse, it;
        public:
                vssut() {cse = it = 0;}
                vssut(float c, float i) {cse = c; it = i;}
                friendvssut operator-(vssut m)
                {
                        m.cse =- m.cse;
                        m.it =- m.it;
                        return m;
                }
                void output()
                {
                        cout<< "\n Real = " <<cse;
                        cout<< "\n Imaginary = " << it;
                }
};
int main()
{
        vssut v1(2.5, 4.5), v2;
        v1.output();
        v2 =- v1;
        cout<< "\n After Operator Overloading \n";
        v2.output();
        return 0;
}
```

OUTPUT

```
Real = 2.5
Imaginary = 4.5
After Operator Overloading
Real = -2.5
Imaginary = -4.5
```

LECTURE-23

Ambiguity (or) Type Conversion

Consider the following statements

int cse;
float it = 2.123456;
cse = it;

Here, the value of it is converted to an integer before it is assigned to cse. This type of conversion is limited to built-in types.

There are three types of data conversion. They are
9. Conversion from basic type to user-defined type (class type)
10. Conversion from class type to basic type
11. Conversion from one class type to another class type

1. Conversion from basic type to user-defined type (class type)

This conversion is automatically processed by the compiler. In this type, the left-hand operand is class type whereas the right-hand operand is basic type.

```
#include<iostream.h>
classvssut
{
        intcse;
        float it;
        public:
                vssut() {cse = 0; it = 0;}
                vssut(float m) {cse = 1; it = m;}
                void output()
                { cout<< "\n cse = " <<cse<< " it = " << it;}
};
int main()
{
        vssut v1;
        v1 = 5;
        v1.output();
        v1 = 1.5;
        v1.output();
        return 0;
}
```

OUTPUT

```
cse = 1 it = 5
cse = 1 it = 1.5
```

2. Conversion from class type to basic type

Here, the programmer specifies the conversion. Alternatively, the compiler is not performing the conversion. This conversion has following constraints:
a. The conversion function does not have any argument-list.

b. There is no return type.
c. It must be a member function.

```
#include<iostream.h>
classvssut
{
        intcse;
        float it;
        public:
                vssut() {cse = 0; it = 0;}
                operatorint() {return cse;}
                operator float() {return it;}
                vssut(float m) { cse = 1; it = m;}
                void input()
                { cout<< "\n cse = " <<cse<< " it = " << it;}
};
int main()
{
        intcse;
        float it;
        vssut v1;
        v1 = 2.5;
        cse = v1;
        it = v1;
        cout<< "\n Value of cse = " <<cse;
        cout<< "\n Value of it = " << it;
        return 0;
}
```

OUTPUT

```
Value of cse = 1
Value of it = 2.5
```

LECTURE-24

Templates

Template is an important feature in C++. It is used for generic programming. A generic program means the functions inside the program should work for all type of data type. It helps to reduce the size of the programs.

Difference between Template and Non Template Function

Standard Template library(STL):

C++ provides a large set of template classes and function that implement many algoriths for data structure programming i.e. vectors, list, queue, stack etc is known as Standard Template Library.

Template classes

To a large degree, a template class is more focused on the algorithmic thought rather than the specific nuances of a single datatype. Templates can be used in conjunction with abstract datatypes in order to allow them to handle any type of data. For example, you could make a templated stack class that can handle a stack of any datatype, rather than having to create a stack class for every different datatype for which you want the stack to function. A single class to do same task for different data type will reduce the line of code and it is easy to handle that class.

Syntax:

template<class a_type> class a_class {...};

Here a_type is not a keyword, it is an identifier that will represent the type of data that needs to be passed to this class to work as a templated class.

a_type a_var;
The above statement will declare an object of a_type class.

References:

1. Ashok N. Kamthane- Object oriented programming with ANSI & Turbo C ++.,Pearso Education.
2. C.Balguru Swamy – C ++, TMH publication.

LECTURE-25

Template

Declaration

Tmplates cann't be declared inside classes or function. They must be global and should not be local. Most of the time the declaration is like

template class<T>
class class_name
{
//class body
}
But in place of

Basic syntax for declaring a template class is

template<class a_type> class a_class {...};

Template functions

When defining a function as a member of a templated class, it is necessary to define it as a templated function:

Syntax:

template<class a_type> void a_class<a_type>::a_function(){...}

Program using template:

```
//~ Using template overloading
#include<iostream>
using namespace std;
template<class T, class P>
class A{
        public:
        void show(T m, T n)
        {
                cout<<"1. inside show\n";
        }
        void show(P n)
        {
                cout<<"2. inside show\n";
        }
        //~ void show(T k)
```

```
	//~ {
		//~ cout<<"3.inside show\n";
	//~ }
};
int main()
{
	A<int, int> obj;
	obj.show(23,34);
	obj.show(4);
}
```

Program 2: The following program will print some error because of function overloading not possible with template

```
//~ template with one of the parameter as built in datatype
#include<iostream>
using namespace std;
template<class T>
class A{
	public:
	void show(T n)
	{
		cout<<n<<"\n";
		cout<<"2. inside show\n";
	}

	void show(int k)
	{
		cout<<k<<"\n";
		cout<<"3. inside show\n";
	}
};
int main()
{
	A <int>obj;
	//~ obj.show(23, 24);
	obj.show(4);
}
```

Program 3:

```
//~ template with one of the parameter as built in datatype
#include<iostream>
using namespace std;
template<class T>
class A{
```

```
        public:
        void show(T n, int m)
        {
                cout<<n<<"\n";
                cout<<"2. inside show\n";
        }

        void show(int k)
        {
                cout<<k<<"\n";
                cout<<"3. inside show\n";
        }
        void show(float k)
        {
                cout<<k<<"\n";
                cout<<"4. inside show\n";
        }
        void show(char k)
        {
                cout<<k<<"\n";
                cout<<"5. inside show\n";
        }

        //~ void show(T k)
        //~ {
                //~ cout<<k<<"\n";
                //~ cout<<"5.inside show\n";
                //~  cannot be overloaded
        //~ }
};
int main()
{
        A<int> obj;
        obj.show(23, 24);
        obj.show(4);
}
```

References:

1. Ashok N. Kamthane- Object oriented programming with ANSI & Turbo C ++.,Pearso Education.
2. C.Balguru Swamy – C ++, TMH publication.

LECTURE-26

Class Templates and Inheritance

It is possible to derive classes from template base class and omit the template features of the derive classes.

Guidelines for templates:

1. Templates are applicable when we want to create type secure class and that can handle different data types with same member functions.
2. The template classes can also be involved in inheritance
3. The template variables also allow us to assign default values
4. The name of the template class is written different situations. While class declararion, it is declared as follows:

class data {};
for member function declaration is as follows
void data<T>:: show(T d){}
where, show() is a member function
Finally, While declaring objects the class name and specific data is specified before object name:
data<int>i1 //object of class data supports interger type
data<float>f1 //object of class data supports float values.

5. All template arguments declared in template argument list should be used for defination of formal arguments. If one of the template arguments is not used, the template will be specious.

Error message will be : wrong number of template arguments

Differnece between Template and Macro

1. Macro are not type safe i.e., a macro definced for interger operation cannot accept float data. They are expanded with no type checking.
2. It is difficult to find error in macros.
3. In case a variable is post-incromented or decremented, the operation is carried out twice.

```
#define max(a) a+ ++a
void  main()
{
int a=4,c;
c=max(a);
cout<<c;
}
```

the macro defined in the above defination is expanded twince. Hence it is a serois limitation of macro. This limitation is overcome using template.

Namespaces

A **namespace** is designed to overcome this difficulty and is used as additional information to differentiate similar functions, classes, variables etc. with the same name available in different libraries. Using namespace, you can define the context in which names are defined. In essence, a namespace defines a scope.

There may be chance that two function are having same name, in that situation the new feature of c++ i.e. namesapce will help in distinguish the context two which it refers to.

STRING

String in C

String is a collection of characters
Every string is terminated by a null character.
The last character of a string is '\0' and the compiler identifies the null character during program execution.

Ex:
char str[6]="Hello";

Here each character of the string occupies single byte in memory as follows
H e l l o '\0'

The various operations with strings such as copying, comparing, concantetion, or replacing requires lot of efforts in 'C' programming. C uses library defined in string.h to carry manipulation.

To make string manipulation easy the ANSI committee added a new class called string. It allows us to define objects of string type and then can be used as built in data type. The string is considered as another container class.

Instead of declaring character array an object of string class is defined. Using member function of string class, string operations such as copying, comparisons, **concatenation** etc. are carried out more easily.

Declaring and Initializing String Objects

In C, we declare sting as given below,

char text[10];
whereas in C++ string is declared as an object. The sting object declarion and initialization can be done at once using constructor in the string class.

```
String str1;                  //Constructor called without argument
String str2("abc");           //Constructor called with one argument
str1=str2;                    //Assignment of two string objects
str1=str1+"is a string";      // Concantation of two string objects
```

String Manipulating Functions

```
append()    // Add one string at the end of another string
assign()    // Assign a specified part of string
at()        //Access the characters located at given location
```

begin() //returns a referenece to the beginning of a string
capacity() //calculates the total elements that can be stored
compare() //compares the two strings
empty() //returns false if the string is not empty, otherwise true
end() // Returns a reference to the termination of string
erase() // Erases the specified character
some more string manipulating functions are find(), insert(), length(), max_size(), replace(), resize(), size(), swap().

Similarly some string manipulating operators are
=, +, +=, ==, !=, <, <=, >, >=, [], <<, >>

Handling String Objects

insert(), replace(), erase() and append() are used to modify the string contents
The member function, insert() is used to insert specified stings into another sting at a given location. It is used in the following form:

s1.insert(2, s2);
Here s1, s2 are two string objects where in the above statement object s2 is used to insert into s1 in location 2.