

Object Oriented Programming Using C++

LECTURE NOTES ON Object Oriented Programming Using C++

**Prepared by
Dr. Subasish Mohapatra**



**Department of Computer Science and Application
College of Engineering and Technology, Bhubaneswar
Biju Patnaik University of Technology, Odisha**

SYLLABUS

PCCS2207 **Object Oriented Programming**

Module I

Introduction to object oriented programming, user defined types, structures, unions, polymorphism, encapsulation. Getting started with C++ syntax, data-type, variables, strings, functions, default values in functions, recursion, namespaces, operators, flow control, arrays and pointers.

Module II

Abstraction mechanism: Classes, private, public, constructors, destructors, member data, member functions, inline function, friend functions, static members, and references.

Inheritance: Class hierarchy, derived classes, single inheritance, multiple, multilevel, hybrid inheritance, role of virtual base class, constructor and destructor execution, base initialization using derived class constructors.

Polymorphism: Binding, Static binding, Dynamic binding, Static polymorphism: Function Overloading, Ambiguity in function overloading, Dynamic polymorphism: Base class pointer, object slicing, late binding, method overriding with virtual functions, pure virtual functions, abstract classes.

Operator Overloading: This pointer, applications of this pointer, Operator function, member and non member operator function, operator overloading, I/O operators.

Exception handling: Try, throw, and catch, exceptions and derived classes, function exception declaration, unexpected exceptions, exception when handling exceptions, resource capture and release.

Module III

Dynamic memory management, new and delete operators, object copying, copy constructor, assignment operator, virtual destructor.

Template: template classes, template functions.

Standard Template Library: Fundamental idea about string, iterators, hashes, iostreams and other types.

Namespaces: user defined namespaces, namespaces provided by library.

Object Oriented Design, design and programming, role of classes.

Text Books:

1. Object Oriented Programming with C++ by E. Balagurusamy, McGraw-Hill Education (India)
2. ANSI and Turbo C++ by Ashoke N. Kamthane, Pearson Education

Reference Books:

1. Big C++ - Wiley India
2. C++: The Complete Reference- Schildt, McGraw-Hill Education (India)
3. C++ and Object Oriented Programming – Jana, PHI Learning.
4. Object Oriented Programming with C++ - Rajiv Sahay, Oxford
5. Mastering C++ - Venugopal, McGraw-Hill Education (India)

CONTENTS

Lecture 01:	Introduction
Lecture 02:	Object Oriented Programming
Lecture 03:	BASIC CONCEPTS OF OBJECTS ORIENTED PROGRAMMING
Lecture 04:	BENEFITS OF OOP
Lecture 05:	Basics of C++
Lecture 06:	Tokens
Lecture 07:	Basic Data types in C++
Lecture 08:	Symbolic Constant
Lecture 09:	Operators
Lecture 10:	Control Structures
Lecture 11:	Functions in C++
Lecture 12:	Function Overloading
Lecture 13:	Class
Lecture 14:	Member Function
Lecture 15:	Nesting of Member function
Lecture 16:	Array with Class
Lecture 17:	Static Data Member
Lecture 18:	Friendly functions
Lecture 19:	Returning Objects
Lecture 20:	Constructors
Lecture 21:	Destructors
Lecture 22 & 23:	Operator Overloading
Lecture 24:	Type Conversion
Lecture 25:	Class to Basic type
Lecture 26:	Inheritance
Lecture 27:	Multilevel Inheritance
Lecture 28:	Hierarchical Inheritance
Lecture 29:	Virtual Base Class
Lecture 30:	Polymorphism
Lecture 31:	Virtual functions
Lecture 32:	Pure Virtual Functions
Lecture 33:	C++ function overriding
Lecture 34:	Exception Handling
Lecture 35:	Array reference out of bound
Lecture 36:	Containership in C++
Lecture 37:	Template
Lecture 38:	Class Template
Lecture 39:	Virtual destructors
Lecture 40:	Managing Console I/O
Lecture 41:	Namespaces
Lecture 42:	New & Delete Operators

Module-1:

LECTURE-1

Introduction:

Programmers write instructions in various programming languages to perform their computation tasks such as:

- (i) Machine level Language
- (ii) Assembly level Language
- (iii) High level Language

Machine level Language :

Machine code or machine language is a set of instructions executed directly by a computer's central processing unit (CPU). Each instruction performs a very specific task, such as a load, a jump, or an ALU operation on a unit of data in a CPU register or memory. Every program directly executed by a CPU is made up of a series of such instructions.

Assembly level Language :

An assembly language (or assembler language) is a low-level programming language for a computer, or other programmable device, in which there is a very strong (generally one-to-one) correspondence between the language and the architecture's machine code instructions. Assembly language is converted into executable machine code by a utility program referred to as an assembler; the conversion process is referred to as assembly, or assembling the code.

High level Language :

High-level language is any programming language that enables development of a program in much simpler programming context and is generally independent of the computer's hardware architecture. High-level language has a higher level of abstraction from the computer, and focuses more on the programming logic rather than the underlying hardware components such as memory addressing and register utilization.

The first high-level programming languages were designed in the 1950s. Now there are dozens of different languages, including Ada , Algol, BASIC, COBOL, C, C++, JAVA, FORTRAN, LISP, Pascal, and Prolog. Such languages are considered high-level because they are closer to human languages and farther from machine languages. In contrast, assembly languages are considered low-level because they are very close to machine languages.

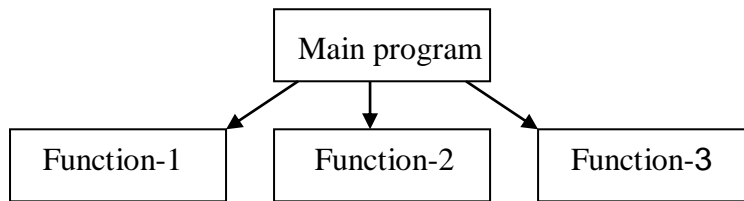
The high-level programming languages are broadly categorized in to two categories:

- (iv) Procedure oriented programming (POP) language.
- (v) Object oriented programming (OOP) language.

Procedure Oriented Programming Language

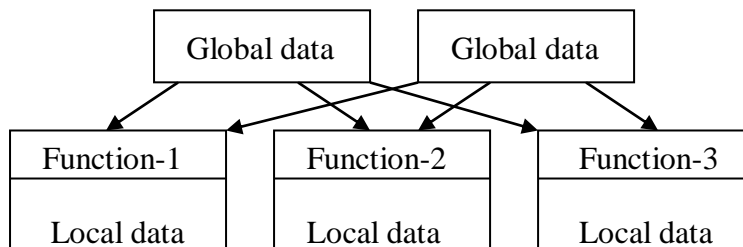
In the procedure oriented approach, the problem is viewed as sequence of things to be done such as reading , calculation and printing.

Procedure oriented programming basically consist of writing a list of instruction or actions for the computer to follow and organizing these instruction into groups known as functions.



The disadvantage of the procedure oriented programming languages is:

1. Global data access
2. It does not model real word problem very well
3. No data hiding



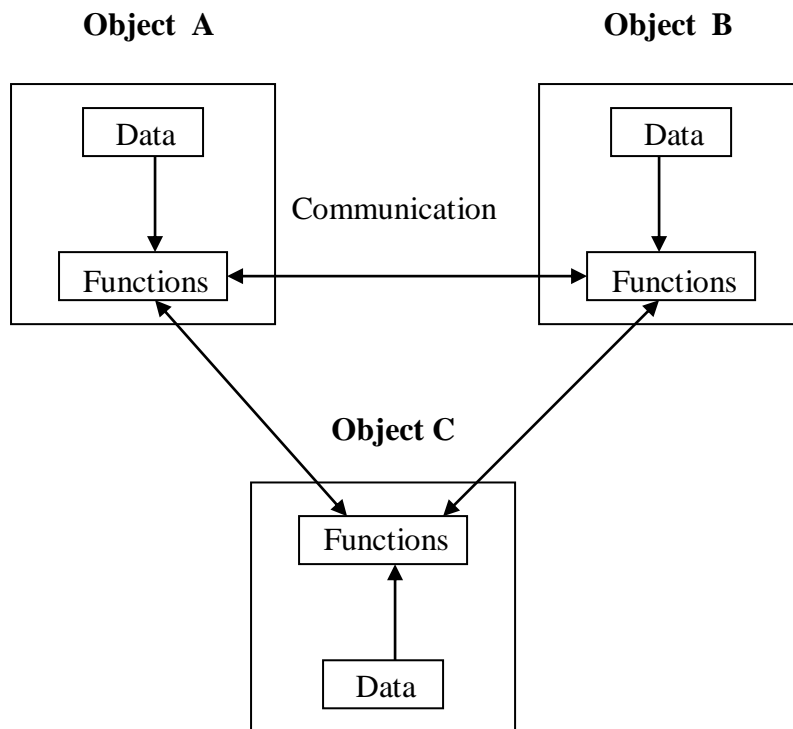
Characteristics of procedure oriented programming:

1. Emphasis is on doing things(algorithm)
2. Large programs are divided into smaller programs known as functions.
3. Most of the functions share global data
4. Data move openly around the system from function to function
5. Function transforms data from one form to another.
6. Employs top-down approach in program design

LECTURE-2

Object Oriented Programing

“Object oriented programming as an approach that provides a way of modularizing programs by creating partitioned memory area for both data and functions that can be used as templates for creating copies of such modules on demand”.



Features of the Object Oriented programming

1. Emphasis is on doing rather than procedure.
2. programs are divided into what are known as objects.
3. Data structures are designed such that they characterize the objects.
4. Functions that operate on the data of an object are tied together in the data structure.
5. Data is hidden and can't be accessed by external functions.
6. Objects may communicate with each other through functions.
7. New data and functions can be easily added.
8. Follows bottom-up approach in program design.

LECTURE-3

BASIC CONCEPTS OF OBJECTS ORIENTED PROGRAMMING

1. Objects
2. Classes
3. Data abstraction and encapsulation
4. Inheritance
5. Polymorphism
6. Dynamic binding
7. Message passing

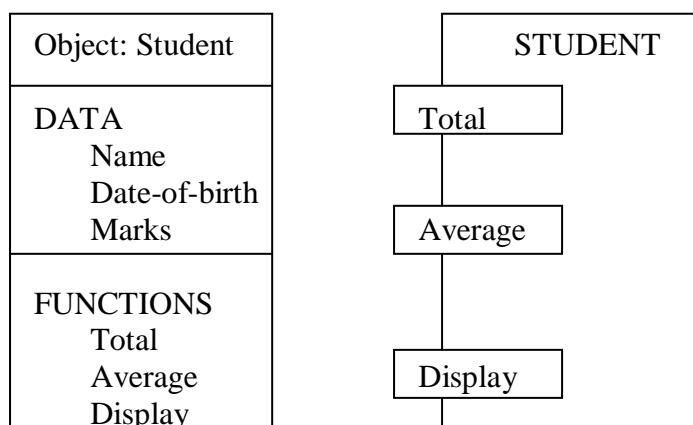
OBJECTS

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 must handle.

The fundamental idea behind object oriented approach is to combine both data and function into a single unit and these units are called objects.

The term objects means a combination of data and program that represent some real word entity. For example: consider an example named Amit; Amit is 25 years old and his salary is 2500. The Amit may be represented in a computer program as an object. The data part of the object would be (name: Amit, age: 25, salary: 2500)

The program part of the object may be collection of programs (retrive of data, change age, change of salary). In general even any user –defined type-such as employee may be used. In the Amit object the name, age and salary are called attributes of the object.



CLASS:

A group of objects that share common properties for data part and some program part are collectively called as class.

In C ++ a class is a new data type that contains member variables and member functions that operate on the variables.

DATA ABSTRACTION :

Abstraction refers to the act of representing essential features without including the background details or explanations. Classes use the concept of abstraction and are defined as size, width and cost and functions to operate on the attributes.

DATA ENCAPSALATION :

The wrapping up of data and function into a single unit (called class) is known as encapsulation. The data is not accessible to the outside world and only those functions which are wrapped in the class can access it. These functions provide the interface between the objects data and the program.

INHERITENCE :

Inheritance is the process by which objects of one class acquire the properties of another class. In the concept of inheritance provides the idea of reusability. This means that we can add additional features to an existing class without modifying it. This is possible by designing a new class which will have the combined features of both the classes.

POLYMORPHISM:

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

A language feature that allows a function or operator to be given more than one definition. The types of the arguments with which the function or operator is called determines which definition will be used.

Overloading may be operator overloading or function overloading.

It is able to express the operation of addition by a single operator say '+'. When this is possible you use the expression $x + y$ to denote the sum of x and y , for many different types of x and y ; integers, float and complex no. You can even define the $+$ operation for two strings to mean the concatenation of the strings.

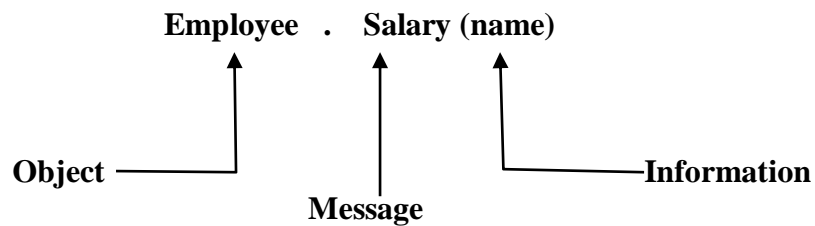
DYNAMIC BINDING :

Binding refers to the linking of a procedure call to the code to be executed in response to the call. Dynamic binding means the code associated with a given procedure call is not known until the time of the call at run-time. It is associated with a polymorphic reference which depends upon the dynamic type of that reference.

MESSAGE PASSING :

An object oriented program consists of a set of objects that communicate with each other.

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 result. Message passing involves specifying the name of the object, the name of the function (message) and information to be sent.



LECTURE- 4

BENEFITS OF OOP:

Oop offers several benefits to both the program designer and the user. Object-oriented contributes to the solution of many problems associated with the development and quality of software products. The principal advantages are :

1. Through inheritance we can eliminate redundant code and extend the use of existing classes.
2. We can build programs from the standard working modules that communicate with one another, rather than having to start writing the code from scratch. This leads to saving of development time and higher productivity.
3. This principle of data hiding helps the programmer to build secure programs that can't be invaded by code in other parts of the program.
4. It is possible to have multiple instances of an object to co-exist with out any interference.
5. It is easy to partition the work in a project based on objects.
6. Object-oriented systems can be easily upgraded from small to large systems.
7. Message passing techniques for communication between objects makes the interface description with external systems much simpler.
8. Software complexity can be easily managed.

APPLICATION OF OOP:

The most popular application of oops up to now, has been in the area of user interface design such as windows. There are hundreds of windowing systems developed using oop techniques.

Real business systems are often much more complex and contain many more objects with complicated attributes and methods. Oop is useful in this type of applications because it can simplify a complex problem. The promising areas for application of oop includes.

1. Real – Time systems.
2. Simulation and modeling
3. Object oriented databases.
4. Hypertext,hypermedia and expertext.
5. Al and expert systems.
6. Neural networks and parallel programming.
7. Dicision support and office automation systems.
8. CIM / CAM / CAD system.

LECTURE-5

Basics of C++

C ++ is an object oriented programming language, C ++ was developed by Jarney Stroustrup at AT & T Bell lab, USA in early eighties. C ++ was developed from c and simula 67 language. C ++ was early called 'C with classes'.

C++ Comments:

C++ introduces a new comment symbol //(double slash). Comments start with a double slash symbol and terminate at the end of line. A comment may start any where in the line and what ever follows till the end of line is ignored. Note that there is no closing symbol.

The double slash comment is basically a single line comment. Multi line comments can be written as follows:

```
// this is an example of  
// c++ program  
// thank you
```

The c comment symbols /**/ are still valid and more suitable for multi line comments.

```
/* this is an example of c++ program */
```

Output Operator:

The statement `cout <<"Hello, world"` displayed the string with in quotes on the screen. The identifier `cout` can be used to display individual characters, strings and even numbers. It is a predefined object that corresponds to the standard output stream. Stream just refers to a flow of data and the standard Output stream normally flows to the screen display. The `cout` object, whose properties are defined in `iostream.h` represents that stream. The insertion operator `<<` also called the 'put to' operator directs the information on its right to the object on its left.

Return Statement:

In C++ `main ()` returns an integer type value to the operating system. Therefore every `main ()` in C++ should end with a `return (0)` statement, otherwise a warning or an error might occur.

Input Operator:

```
The statement  
cin>> number 1;
```

is an input statement and causes. The program to wait for the user to type in a number. The number keyed in is placed in the variable `number1`. The identifier **cin** is a predefined object in C++ that corresponds to the standard input stream. Here this stream represents the key board.

The operator `>>` is known as get from operator. It extracts value from the keyboard and assigns it to the variable on its right.

Cascading Of I/O Operator:

```
cout<<"sum="<<sum<<"\n";
cout<<"sum="<<sum<<"\n"<<"average="<<average<<"\n";
cin>>number1>>number2;
```

Structure Of A Program :

Probably the best way to start learning a programming language is by writing a program. Therefore, here is our first program:

// my first program in C++

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

```
int main ()
{
    cout << "Hello World!";
    return 0;
}
```

Output:-Hello World!

The first panel shows the source code for our first program. The second one shows the result of the program once compiled and executed. The way to edit and compile a program depends on the compiler you are using. Depending on whether it has a Development Interface or not and on its version. Consult the compilers section and the manual or help included with your compiler if you have doubts on how to compile a C++ console program.

The previous program is the typical program that programmer apprentices write for the first time, and its result is the printing on screen of the "Hello World!" sentence. It is one of the simplest programs that can be written in C++, but it already contains the fundamental components that every C++ program has. We are going to look line by line at the code we have just written:

// my first program in C++

This is a comment line. All lines beginning with two slash signs (//) are considered comments and do not have any effect on the behavior of the program. The programmer can use them to include short explanations or observations within the source code itself. In this case, the line is a brief description of what our program is.

#include <iostream>

Lines beginning with a hash sign (#) are directives for the preprocessor. They are not regular code lines with expressions but indications for the compiler's preprocessor. In this case the directive `#include<iostream>` tells the preprocessor to include the `iostream` standard file. This specific file (`iostream`) includes the declarations of the basic standard input-output library in C++, and it is included because its functionality is going to be used later in the program.

using namespace std;

All the elements of the standard C++ library are declared within what is called a namespace, the namespace with the name *std*. So in order to access its functionality we declare with this expression that we will be using these entities. This line is very frequent in C++ programs that use the standard library, and in fact it will be included in most of the source codes included in these tutorials.

int main ()

This line corresponds to the beginning of the definition of the main function. The main function is the point by where all C++ programs start their execution, independently of its location within the source code. It does not matter whether there are other functions with other names defined before or after it – the instructions contained within this function's definition will always be the first ones to be

executed in any C++ program. For that same reason, it is essential that all C++ programs have a main function.

The word main is followed in the code by a pair of parentheses (). That is because it is a function declaration: In C++, what differentiates a function declaration from other types of expressions are these parentheses that follow its name. Optionally, these parentheses may enclose a list of parameters within them.

Right after these parentheses we can find the body of the main function enclosed in braces {}. What is contained within these braces is what the function does when it is executed.

cout << "Hello World!";

This line is a C++ statement. A statement is a simple or compound expression that can actually produce some effect. In fact, this statement performs the only action that generates a visible effect in our first program.

cout represents the standard output stream in C++, and the meaning of the entire statement is to insert a sequence of characters (in this case the Hello World sequence of characters) into the standard output stream (which usually is the screen).

cout is declared in the iostream standard file within the std namespace, so that's why we needed to include that specific file and to declare that we were going to use this specific namespace earlier in our code.

Notice that the statement ends with a semicolon character (;). This character is used to mark the end of the statement and in fact it must be included at the end of all expression statements in all C++ programs (one of the most common syntax errors is indeed to forget to include some semicolon after a statement).

return 0;

The return statement causes the main function to finish. return may be followed by a return code (in our example is followed by the return code 0). A return code of 0 for the main function is generally interpreted as the program worked as expected without any errors during its execution. This is the most usual way to end a C++ console program.

You may have noticed that not all the lines of this program perform actions when the code is executed. There were lines containing only comments (those beginning by //). There were lines with directives for the compiler's preprocessor (those beginning by #). Then there were lines that began the declaration of a function (in this case, the main function) and, finally lines with statements (like the insertion into cout), which were all included within the block delimited by the braces {} of the main function.

The program has been structured in different lines in order to be more readable, but in C++, we do not have strict rules on how to separate instructions in different lines. For example, instead of

```
int main ()
{
    cout << " Hello World!";
    return 0;
}
```

We could have written:

```
int main ()
{
cout << "Hello World!";
return 0;
}
```

All in just one line and this would have had exactly the same meaning as the previous code.

In C++, the separation between statements is specified with an ending semicolon (;) at the end of each one, so the separation in different code lines does not matter at all for this purpose. We can write many statements per line or write a single statement that takes many code lines. The division of

code in different lines serves only to make it more legible and schematic for the humans that may read it.

Let us add an additional instruction to our first program:

```
// my second program in C++
#include <iostream>
using namespace std;

int main ()
{
    cout << "Hello World! ";
    cout << "I'm a C++ program";
    return 0;
}
```

Output:-Hello World! I'm a C++ program

In this case, we performed two insertions into cout in two different statements. Once again, the separation in different lines of code has been done just to give greater readability to the program, since main could have been perfectly valid defined this way:

```
int main ()
{
    cout << " Hello World! ";
    cout << " I'm a C++ program ";
    return 0;
}
```

We were also free to divide the code into more lines if we considered it more convenient:

```
int main ()
{
    cout << "Hello World!";
    cout << "I'm a C++ program";
    return 0;
}
```

And the result would again have been exactly the same as in the previous examples.

Preprocessor directives (those that begin by #) are out of this general rule since they are not statements. They are lines read and processed by the preprocessor and do not produce any code by themselves. Preprocessor directives must be specified in their own line and do not have to end with a semicolon (;).

STRUCTURE OF C++ PROGRAM

- **Include files**
- **Class declaration**
- **Class functions, definition**
- **Main function program**

Example :-

```
# include<iostream.h>

class person
```

```

{
char name[30];
int age;
public:
    void getdata(void);
    void display(void);
};

void person :: getdata ( void )
{
    cout<<"enter name";
    cin>>name;
    cout<<"enter age";
    cin>>age;
}

void display()
{
    cout<<"\n name:"<<name;
    cout<<"\n age:"<<age;
}

int main( )
{

person p;
p.getdata();
p.display();
return(0);

}

```


LECTURE-6

TOKENS:

The smallest individual units in program are known as **tokens**. C++ has the following tokens.

- i. Keywords
- ii. Identifiers
- iii. Constants
- iv. Strings
- v. Operators

KEYWORDS:

The keywords implement specific C++ language feature. They are explicitly reserved identifiers and can't be used as names for the program variables or other user defined program elements. The keywords not found in ANSI C are shown in red letter.

C++ KEYWORDS:

Asm	double	new	switch
Auto	else	operator	template
Break	enum	private	this
Case	extern	protected	throw
Catch	float	public	try
Char	for	register	typedef
Class	friend	return	union
Const	goto	short	unsigned
Continue	if	signed	virtual
Default	inline	sizeof	void
Delete	long	struet	while

IDENTIFIERS:

Identifiers refers to the name of variable , functions, array, class etc. created by programmer. Each language has its own rule for naming the identifiers.

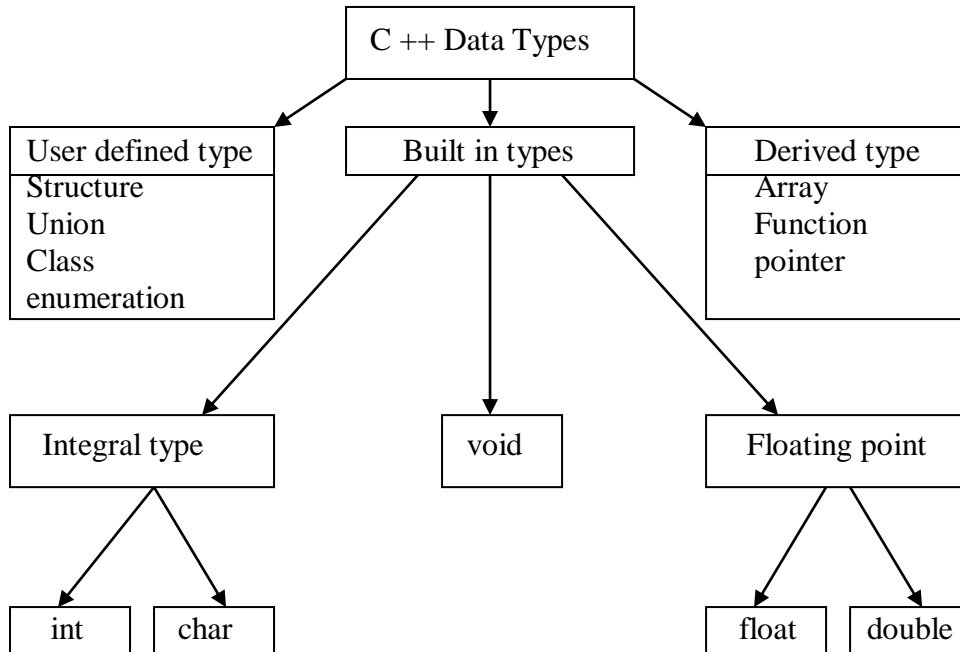
The following rules are common for both C and C++.

1. Only alphabetic chars, digits and under score are permitted.
2. The name can't start with a digit.
3. Upper case and lower case letters are distinct.
4. A declared keyword can't be used as a variable name.

In ANSI C the maximum length of a variable is 32 chars but in c++ there is no bar.

Lecture-7

BASIC DATA TYPES IN C++



Both C and C++ compilers support all the built in types. With the exception of void the basic datatypes may have several modifiers preceding them to serve the needs of various situations. The modifiers signed, unsigned, long and short may applied to character and integer basic data types. However the modifier long may also be applied to double.

Data types in C++ can be classified under various categories.

<u>TYPE</u>	<u>BYTES</u>	<u>RANGE</u>
char	1	-128 to – 127
unsigned	1	0 to 265
sgned char	1	-128 to 127
int	2	-32768 to 32768
unsigned int	2	0 to 65535
singed int	2	-32768 to 32768
short int	2	-32768 to 32768

long int	4	-2147483648 to 2147483648
signed long int	4	-2147483648 to 2147483648
unsigned long int	4	0 to 4294967295
float	4	3.4E-38 to 3.4E+38
double	8	1.7E -308 to 1.7E +308
long double	10	3.4E-4932 to 1.1E+ 4932

The type void normally used for:

- 1) To specify the return type of function when it is not returning any value.
- 2) To indicate an empty argument list to a function.

Example:

Void function(void);

Another interesting use of void is in the declaration of generic pointer

Example:

Void *gp;

Assigning any pointer type to a void pointer without using a cast is allowed in both C and ANSI C. In ANSI C we can also assign a void pointer to a non-void pointer without using a cast to non void pointer type. This is not allowed in C ++.

Example:

void *ptr1;

void *ptr2;

Are valid statement in ANSI C but not in C++. We need to use a cast operator.

ptr2=(char *) ptr1;

USER DEFINED DATA TYPES:

STRUCTURES AND CLASSES

We have used user defined data types such as struct, and union in C. While these more features have been added to make them suitable for object oriented programming. C++ also permits us to define

another user defined data type known as class which can be used just like any other basic data type to declare a variable. The class variables are known as objects, which are the central focus of oops.

ENUMERATED DATA TYPE:

An enumerated data type is another user defined type which provides a way for attaching names to number, these by increasing comprehensibility of the code. The enum keyword automatically enumerates a list of words by assigning them values 0,1,2 and soon. This facility provides an alternative means for creating symbolic.

Example:

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

```
enum colour{red,blue,green,yellow}
```

```
enum position {off,on}
```

The enumerated data types differ slightly in C++ when compared with ANSI C. In C++, the tag names shape, colour, and position become new type names. That means we can declare new variables using the tag names.

Example:

```
Shape ellipse;//ellipse is of type shape
```

```
colour background ; // back ground is of type colour
```

ANSI C defines the types of enums to be ints. In C++,each enumerated data type retains its own separate type. This means that C++ does not allow an int value to be automatically converted to an enum.

Example:

```
colour background =blue; //vaid
```

```
colour background =7; //error in c++
```

```
colour background =(colour) 7;//ok
```

How ever an enumerated value can be used in place of an int value.

Example:

```
int c=red ;//valid, colour type promoted to int
```

By default, the enumerators are assigned integer values starting with 0 for the first enumerator, 1 for the second and so on. We can also write

```
enum color {red, blue=4,green=8};
```

```
enum color {red=5,blue,green};
```

C++ also permits the creation of anonymous enums (i.e, enums without tag names)

Example:

```
enum{off,on};
```

Here off is 0 and on is 1. these constants may be referenced in the same manner as regular constants.

Example:

```
int switch-1=off;
```

```
int switch-2=on;
```

ANSI C permits an enum defined with in a structure or a class, but the enum is globally visible. In C++ an enum defined with in a class is local to that class.

LECTURE-8

SYMBOLIC CONSTANT:

There are two ways of creating symbolic constants in c++.

1. using the qualifier const.
2. defining a set of integer constants using enum keywords.

In both C and C++, any value declared as const can't be modified by the program in any way. In C++, we can use const in a constant expression. Such as

```
const int size = 10 ;  
char name (size) ;
```

This would be illegal in C. const allows us to create typed constants instead of having to use #define to create constants that have no type information.

```
const size=10;
```

Means

```
const int size =10;
```

C++ requires a const to be initialized. ANSI C does not require an initializer, if none is given, it initializes the const to 0.

In C++ const values are local and in ANSI C const values are global .However they can be made local made local by declaring them as static .In C++ if we want to make const value as global then declare as extern storage class.

Ex: external const total=100;
of naming integer constants is as follows:-

Another method

```
enum {x,y,z};
```

DECLARATION OF VARIABLES:

In ANSIC C all the variable which is to be used in programs must be declared at the beginning of the program .But in C++ we can declare the variables any where in the program where it requires .This makes the program much easier to write and reduces the errors that may be caused by having to scan back and forth. It also makes the program easier to understand because the variables are declared in the context of their use.

Example:

```
main()  
{  
    float x,average;  
    float sum=0;
```

```

    for(int i=1;i<5;i++)
    {
        cin>>x;
        sum=sum+x;
    }
    float average;
    average=sum/x;
    cout<<average;
}

```

REFERENCE VARIABLES:

C++ interfaces a new kind of variable known as the reference variable. A reference variable provides an alias (alternative name) for a previously defined variable. For example, if we make the variable sum a reference to the variable total, then sum and total can be used interchangeably to represent the variable.

A reference variable is created as follows:

Syntax: Datatype & reference-name=variable name;

Example:

```

float total=1500;
float &sum=total;

```

Here sum is the alternative name for variable total, both the variables refer to the same data object in the memory.

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

Note that C++ assigns additional meaning to the symbol & here & is not an address operator. The notation float & means reference to float.

Example:

```

int n[10];
int &x=n[10];
char &a='n';

```


LECTURE-9

OPERATORS IN C++:

C++ has a rich set of operators. All C operators are valid in C++ also. In addition, C++ introduces some new operators.

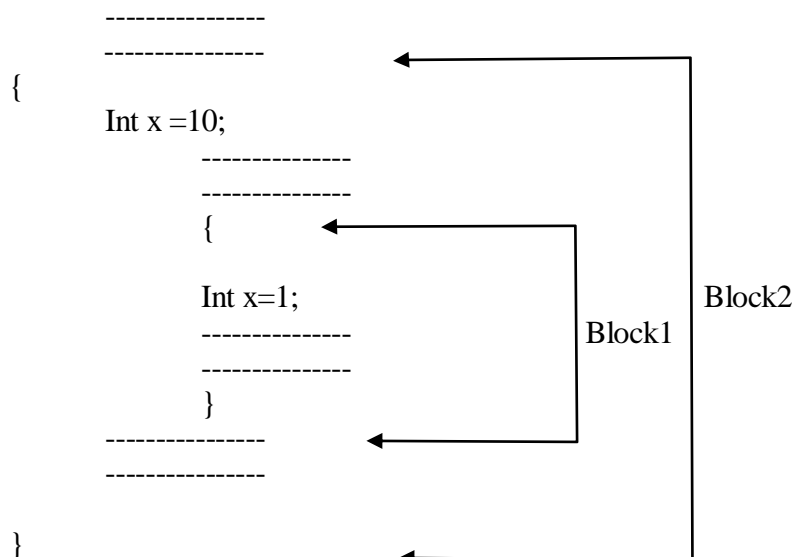
<<	insertion operator
>>	extraction operator
::	scope resolution operator
::*	pointer to member declarator
*	pointer to member operator
.*	pointer to member operator
Delete	memory release operator
Endl	line feed operator
New	memory allocation operator
Setw	field width operator

SCOPE RESOLUTION OPERATOR:

Like C, C++ is also a block-structured language. Block-structured language. Blocks and scopes can be used in constructing programs. We know same variables can be declared in different blocks because the variables declared in blocks are local to that function.

Blocks in C++ are often nested.

Example:



Block2 contained in block 1. Note that declaration in an inner block hides a declaration of the same variable in an outer block and therefore each declaration of x causes it to refer to a different data object. Within the inner block the variable x will refer to the data object declared there in.

In C, the global version of a variable can't be accessed from within the inner block. C++ resolves this problem by introducing a new operator :: called the scope resolution operator. This can be used to uncover a hidden variable.

Syntax: :: variable -name;

Example:

```
#include <iostream.h>
int m=10;
main()
{
    int m=20;
    {
        int k=m;
        int m=30;
        cout<<"we are in inner block";
        cout<<"k="<<k<<endl;
        cout<<"m="<<m<<endl;
        cout<<":: m="<<:: m<<endl;
    }
    cout<<"\n we are in outer block \n";
    cout<<"m="<<m<<endl;
    cout<<":: m="<<:: m<<endl;
}
```

Memory Management Operator

C uses malloc and calloc functions to allocate memory dynamically at run time. Similarly it uses the functions Free() to free dynamically allocated memory. We use dynamic allocation techniques when it is not known in advance how much of memory space is needed.

C++ also supports those functions; it also defines two unary operators new and delete that perform the task of allocating and freeing the memory in a better and easier way.

The new operator can be used to create objects of any type. **Syntax:** pointer-

variable =new datatype;

Example:

```
p=new int; q=new int;
```

Where p is a pointer of type int and q is a pointer of type float.

```
int *p=new int;
```

```
float *p=new float;
```

Subsequently, the statements

```
*p=25;
```

`*q=7.5;`

Assign 25 to the newly created int object and 7.5 to the float object. We can also initialize the memory using the new operator.

Syntax:

```
int *p=new int(25);  
float *q=new float(7.5);
```

new can be used to create a memory space for any data type including user defined such as arrays, structures, and classes. The general form for a one-dimensional array is:

pointer-variable =new data types [size];

creates a memory space for an array of 10 integers.

If a data object is no longer needed, it is destroyed to release the memory space for reuse.

Syntax: delete pointer-variable;

Example:

```
delete p;  
delete q;
```

If we want to free a dynamically allocated array, we must use the following form of delete.

```
delete [size] pointer-variable;  
or
```

```
delete [ ] pointer variable;
```

MANIPULATORS:

Manipulators are operators that are used to format the data display. The most commonly manipulators are endl and setw.

The endl manipulator, when used in an output statement, causes a line feed to be inserted (just like \n).

Example:

```
cout<<"m="<<m<<endl;  
cout<<"n="<<n<<endl;  
cout<<"p="<<p<<endl;
```

If we assume the values of the variables as 2597, 14 and 175 respectively

```
m=2597; n=14;  
p=175
```

It was wanted to print all nos in right justified way use setw which specifies a common field width for all the nos.

```
Example: cout<<setw(5)<<sum<<endl;  
cout<<setw(10)<<"basic"<<setw(10)<<basic<<endl;  
cout<<setw(10)<<"allowance"<<setw(10)<<allowance<<endl;  
cout<<setw(10)<<"total="<<setw(10)<<total;
```

LECTURE-10

CONTROL STRUCTURES:

Like c,c++, supports all the basic control structures and implements them various control statements.

The if statement:

The if statement is implemented in two forms:

1. simple if statement
2. if... else statement

Simple if statement:

if (condition)

{

Action;

}

If.. else statement

If (condition)

Statment1

Else

Statement2

The switch statement

This is a multiple-branching statement where, based on a condition, the control is transferred to one of the many possible points;

Switch(expr)

{

case 1:

action1;

break;

case 2:

action2;

break;

..

..

default:

message

}

The while statement:

Syn:

While(condition)

{

Stements

}

The do-while statement:

Syn:

do

{

Statements

} while(condition);

The for loop:

for(expression1;expression2;expression3)

{

Statements;

Statements;

}

LECTURE-11

FUNCTION IN C++:

The main() Function :

ANSI does not specify any return type for the main () function which is the starting point for the execution of a program . The definition of main() is :-

```
main()
{
//main program statements
}
```

This is property valid because the main () in ANSI C does not return any value. In C++, the main () returns a value of type int to the operating system. The functions that have a return value should use the return statement for terminating. The main () function in C++ is therefore defined as follows.

```
int main( )
{
-----
-----
return(0)
}
```

Since the return type of functions is int by default, the key word int in the main() header is optional.

INLINE FUNCTION:

To eliminate the cost of calls to small functions C++ proposes a new feature called inline function. An inline function is a function that is expanded inline when it is invoked .That is the compiler replaces the function call with the corresponding function code.

The inline functions are defined as follows:-

```
inline function-header
{
function body;
}
```

Example: inline double cube (double a)

```
              {
              return(a*a*a);
              }
```

The above inline function can be invoked by statements like

```
c=cube(3.0);
d=cube(2.5+1.5);
```

remember that the inline keyword merely sends a request, not a command to the compiler. The compiler may ignore this request if the function definition is too long or too complicated and compile the function as a normal function.

Some of the situations where inline expansion may not work are:

1. For functions returning values if a loop, a switch or a go to exists.

2. for function s not returning values, if a return statement exists.
3. if functions contain static variables.
4. if inline functions are recursive,.

Example:

```
#include<iostream.h>
#include<stdio.h>
inline float mul(float x, float y)
{
    return(x*y);
}
inline double div(double p,double q)
{
    return(p/q);
}

main( )
{
    float a=12.345;
    float b=9.82;
    cout<<mul(a,b)<<endl;
    cout<<div (a,b)<<endl;
}
```

output:-

121.227898
1.257128

DEFAULT ARGUMENT:-

C++ allows us to call a function with out specifying all its arguments. In such cases, the function assigns a default value to the parameter which does not have a matching arguments in the function call. Default values are specified when the function is declared. The compiler looks at the prototype to see how many arguments a function uses and alerts the program for possible default values.

Example: float amount (float principle, int period ,float rate=0.15);

The default value is specified in a manner syntactically similar to a variable initialization. The above prototype declares a default value of 0.15 to the argument rate. A subsequent function call like

value=amount(5000,7); //one argument missing

passes the value of 5000 to principle and 7 to period and then lets the function, use default value of 0.15 for rate.

The call:- value=amount(5000,5,0.12);

//no missing argument passes an explicit value of 0.12 rate.

One important point to note is that only the trailing arguments can have default values. That is, we must add default from right to left. We cannot provide a default to a particular argument in the middle of an argument list.

Example:- int mul(int i, int j=5,int k=10); //illegal
int mul(int i=0,int j,int k=10); //illegal
int mul(int i=5,int j); //illegal
int mul(int i=2,int j=5,int k=10); //illegal

Default arguments are useful in situation whose some arguments always have the same value. For example, bank interest may retain the same for all customers for a particular period of deposit.

Example:

```
#include<iostream.h>
#include<stdio.h>
mainQ
{
float amount;
float value(float p,int n,float r=0.15);
void printline(char ch='*',int len=40);
printline( );
amount=value(5000.00,5);
cout<<"\n final value="<<amount<<endl;
printline('=');
//function definitions
float value (float p,int n, float r)
{
    float si;
    si=(p*n*r)/100;
    return(si);
}
void printline (char ch,int len)
{
    for(inti=1;i<=len;i++)
        cout<<ch<<endl;
}
}
output:-
*****
final value=10056.71613
=====
```

Advantage of providing the default arguments are:

1. We can use default arguments to add new parameters to the existing functions.
2. Default arguments can be used to combine similar functions into one.

CONST ARGUMENT:-

In C++, an argument to a function can be declared as const as shown below.

```
int strlen(const char *p);
int length(const string &s);
```

The qualifier const tells the compiler that the function should not modify the argument. The compiler will generate an error when this condition is violated. This type of declaration is significant only when we pass arguments by reference or pointers.

LECTURE-12

FUNCTION OVERLOADING:

Overloading refers to the use of the same thing for different purposes . C++ also permits overloading functions .This means that we can use the same function name to creates functions that perform a variety of different tasks. This is known as function polymorphism in oops.

Using the concepts of function overloading , a family of functions with one function name but with different argument lists in the functions call .The correct function to be invoked is determined by checking the number and type of the arguments but not on the function type.

For example an overloaded add() function handles different types of data as shown below.

//Declaration

```
int add(int a, int b); //prototype 1  
int add (int a, int b, int c); //prototype 2  
double add(double x, double y); //prototype 3  
double add(double p , double q); //prototype 4
```

//function call

```
cout<<add(5,10); //uses prototype 1  
cout<<add(15,10.0); //uses prototype 4  
cout<<add(12.5,7.5); //uses prototype 3  
cout<<add(5,10,15); //uses prototype 2  
cout<<add(0.75,5); //uses prototype 5
```

A function call first matches the prototype having the same no and type of arguments and then calls the appropriate function for execution.

The function selection invokes the following steps:-

- a) The compiler first tries to find an exact match in which the types of actual arguments are the same and use that function .
- b) If an exact match is not found the compiler uses the integral promotions to the actual arguments such as :

char to int
float to double
to find a match

c)When either of them fails ,the compiler tries to use the built in conversions to the actual arguments and then uses the function whose match is unique . If the conversion is possible to have multiple matches, then the compiler will give error message.

Example:

long square (long n);
double square(double x);

A function call such as :- square(10)

Will cause an error because int argument can be converted to either long or double .There by creating an ambiguous situation as to which version of square()should be used.

PROGRAM

```
#include<iostream.h>
int volume(double,int);
double volume( double , int );
double volume(longint ,int ,int);
main( )
{
    cout<<volume(10)<<endl;
    cout<<volume(10)<<endl; cout<<volume(10)<<endl;
}
int volume( ini s)
{
    return (s*s*s); //cube
}
double volume( double r, int h)
{
    return(3.1416*r*r*h); //cylinder
}
long volume (longint l, int b, int h)
{
    return(1*b*h); //cylinder
}
```

output:- 1000
157.2595
112500

Module-2:

LECTURE-13

CLASS:-

Class is a group of objects that share common properties and relationships .In C++, a class is a new data type that contains member variables and member functions that operates on the variables. A class is defined with the keyword class. It allows the data to be hidden, if necessary from external use. When we defining a class, we are creating a new abstract data type that can be treated like any other built in data type.

Generally a class specification has two parts:-

- a) Class declaration
- b) Class function definition

the class declaration describes the type and scope of its members. The class function definition describes how the class functions are implemented.

Syntax:-

```
class class-name
{
    private:
        variable declarations;
        function declaration ;
    public:
        variable declarations;
        function declaration;
};
```

The members that have been declared as private can be accessed only from with in the class. On the other hand , public members can be accessed from outside the class also. The data hiding is the key feature of oops. The use of keywords private is optional by default, the members of a class are private.

The variables declared inside the class are known as data members and the functions are known as members mid the functions. Only the member functions can have access to the private data members and private functions. However, the public members can be accessed from the outside the class. The binding of data and functions together into a single class type variable is referred to as encapsulation.

Syntax:-

```
class item
{
    int member;
    float cost;
    public:
        void getldata (int a ,float b);
        void putdata (void);
```

The class item contains two data members and two function members, the data members are private by default while both the functions are public by declaration. The function getdata() can be used to assign values to the member variables member and cost, and putdata() for displaying their values . These functions provide the only access to the data members from outside the class.

CREATING OBJECTS:

Once a class has been declared we can create variables of that type by using the class name.

Example:

item x;

creates a variable x of type item. In C++, the class variables are known as objects. Therefore x is called an object of type item.

item x, y, z also possible.

```
class item
```

```
{
```

```
-----
```

```
-----
```

```
-----
```

```
}x,y,z;
```

would create the objects x, y, z of type item.

ACCESSING CLASS MEMBER:

The private data of a class can be accessed only through the member functions of that class. The main() cannot contain statements that access the number and cost directly.

Syntax:

object name.function-name(actual arguments);

Example:- x.getdata(100,75.5);

It assigns value 100 to number, and 75.5 to cost of the object x by implementing the getdata() function.

similarly the statement

x.putdata (); //would display the values of data members.

x.number = 100 is illegal. Although x is an object of the type item to which number belongs, the number can be accessed only through a member function and not by the object directly.

Example:

```
class xyz
```

```
{
```

```
    int x;
```

```
    int y;
```

```
public:
```

```
    int z;
```

```
};
```

```
-----
```

```
-----
```

```
xyz p;
```

```
p.x=0;
```

```
p.z=10;
```

error . x is private

ok ,z is public

LECTURE-14

DEFINING MEMBER FUNCTION:

Member can be defined in two places

- Outside the class definition
- Inside the class function

OUTSIDE THE CLASS DEFINATION:

Member function that are declared inside a class have to be defined separately outside the class. Their definition are very much like the normal functions.

An important difference between a member function and a normal function is that a member function incorporates a membership. Identify label in the header. The 'label' tells the compiler which class the function belongs to.

Syntax:

```
return type class-name::function-name(argument declaration )
{
    function-body
}
```

The member ship label class-name :: tells the compiler that the function function - name belongs to the class class-name . That is the scope of the function is restricted to the class-name specified in the header line. The :: symbol is called scope resolution operator.

Example:

```
void item :: getdata (int a , float b )
{
    number=a;
    cost=b;
}
void item :: putdata ( void)
{
    cout<<"number="<<number<<endl;
    cout<<"cost="<<cost<<endl;
}
```

The member function have some special characteristics that are often used in the program development.

- Several different classes can use the same function name. The "membership label" will resolve their scope, member functions can access the private data of the class .A non member function can't do so.
- A member function can call another member function directly, without using the dot operator.

INSIDE THE CLASS DEFINITION:

Another method of defining a member function is to replace the function declaration by the actual function definition inside the class .

Example:

```
class item
{
    Intnumber;
    float cost;
public:
    void getdata (int a ,float b);
    void putdata(void)
    {
        cout<<number<<endl; cout<<cost<<endl;
    }
};
```

A C++ PROGRAM WITH CLASS:

```
#include< iostream. h>
class item
{
    int number;
    float cost;
public:
    void getdata ( int a , float b);
    void putdala ( void)
    {
        cout<<"number:"<<number<<endl;
        cout<<"cost : "<<cost<<endl;
    }
};

void item :: getdata (int a , float b)
{
    number=a;
    cost=b;
}

main ( )
{
    item x;
    cout<<"\nobjectx"<<endl;
    x. getdata( 100,299.95);
    x .putdata();
    item y;
    cout<<"\n object y"<<endl;
    y. getdata(200,175.5);
    y. putdata();
}
```

Output: **object x**
 number 100

cost=299.950012
object -4
cost=175.5

Q.

Write a simple program using class in C++ to input subject mark and prints it.
ans:

```
class marks
{
    private :
        int ml,m2;
    public:
        void getdata();
        void displaydata();
};
void marks::getdata()
{
    cout<<"enter 1st subject mark:";
    cin>>ml;
    cout<<"enter 2nd subject mark:";
    cin>>m2;
}
void marks::displaydata()
{
    cout<<"1st subject mark:"<<ml<<endl ;
    cout<<"2nd subject mark:"<<m2;
}
void main()
{
    clrscr();
    marks x;
    x.getdata();
    x.displaydata();
}
```


LECTURE-15

NESTING OF MEMBER FUNCTION:

A member function can be called by using its name inside another member function of the same class. This is known as nesting of member functions.

```
#include <iostream.h>
class set
{
    int m,n;
    public:
        void input(void);
        void display (void);
        void largest(void);
};
int set::largest (void)
{
    if(m>n)
        return m;
    else
        return n;
}
void set::input(void)
{
    cout<<"input values of m and n:";
    cin>>m>>n;
}
void set::display(void)
{
    cout<<"largestvalue="<<largest()<<"\n";
}
void main()
{
    set A;
    A.input( );
    A.display( );
}
```

output:

Input values of m and n:

3017

largest value= 30

Private member functions:

Although it is a normal practice to place all the data items in a private section and all the functions in public, some situations may require contain functions to be hidden from the outside calls. Tasks such as deleting an account in a customer file or providing increment to an employee are events of serious consequences and therefore the functions handling such tasks should have restricted access. We can place these functions in the private section.

A private member function can only be called by another function that is a member of its class. Even an object can not invoke a private function using the dot operator.

Class sample

```
{  
    int m;  
    void read (void);  
    void write (void);  
};
```

if si is an object of sample, then

```
s.read();
```

is illegal. However the function read() can be called by the function update () to update the value of m.

```
void sample :: update(void)  
{  
    read( );  
}
```

```

#include<iostream.h>

class part
{
private:
    int  modelnum,partnum;
    float cost;
public:
    void setpart ( int mn, int pn ,float c)
    {
        modelmim=mn;
        partnum=pn;
        cost=e;
    }
    void showpart ( )
    {
        Cout<<endl<<"model:"<<modelnum<<endl;
        Cout<<"num:"<< partnum <<endl
        Cout<<"cost:"<<"$"<<cost;
    }
};

void main()
{
    part pl,p2;
    p1.setpart(644,73,217.55);
    p2.setpart(567,89,789.55);
    pl.showpart();
    pl.showpart();
}

```

output:- model:644

num:73

cost: \$217550003

model: 567

num:89

cost: \$759.549988

```

#include<iostream.h>
class distance
{
private:
    int feet;
    float inches;
public:
    void setdist ( int ft, float in)
    {
        feet=ft;
        inches=in;
    }
    void getdist()
    {
        cout<<"enter feet:";
        cin>>feet;
        cout<<"enter inches:";
        cin>>inches;
    }
    void showdist()
    {
        cout<< feet<<" "inches<<endl;
    }
};
void main( )
{
    distance d1,d2;
    d1.setdist(11,6.25);
    d2.getdata();
    cout<<endl<<"dist:"<<d1.showdist();
    cout<<"\n"<<"dist2:";
    d2.showdist();
}

```

output:- enter feet: 12
 enter inches: 6.25
 dist 1:"11' - 6.1.5"
 dist 2: 12' - 6.25"

LECTURE-16

ARRAY WITH CLASSES:

```
#include<iostream.h>
#include<conio.h>
class employee
{
private:
    char name[20];
    int age,sal;
public:
    void getdata();
    void putdata();
};

void employee :: getdata ()
{
    cout<<"enter name :";
    cin>>name;
    cout<<"enter age :";
    cin>>age;
    cout<<"enter salary:";
    cin>>sal;
    return(0);
}

void employee :: putdata ( )
{
    cout<<name <<endl;
    cout<<age<<endl;
    cout<<sal<<endl;
    return(0);
}

int main()
{
```

```

employee emp[5]:
for( int i=0;i<5;i++)
{
emp[i].getdata();
}
cout<<endl;
for(i=0;i<5;i++)
{
emp[i].putdata();
}
getch();
return(0);
}

```

ARRAY OF OBJECTS:-

```

#include<iostream.h>
#include<conio.h>
class emp
{
private:
    char name[20];
    int age,sal;
public:
    void getdata( );
    void putdata( );
};

void emp :: getdata( )
{
    cout<<"enter empname": .
    cin>>name;
    cout<<"enter age:"<<endl;
    cin>>age;
    cout<<"enter salun :";
}

```

```

        cin>>sal;

    }

void emp :: putdata ()
{
    cout<<"emp name:"<<name<<endl;
    cout<<"emp age:"<<age<<endl;
    cout<<"emp salary:"<<sal;
}

void main()
{
    emp foreman[5];
    emp engineer[5];
    for(int i=0;i<5;i++)
    {
        cout<<" for foreman:";
        foreman[i] . getdata();
    }
    cout<<endl;
    for(i=0;i<5;i++)
    {
        Foreman[i].putdata(); .
    }
    for(int i=0;i<5;i++)
    {
        cout<<" for engineer:";
        ingineer[i].getdata();
    }
    for(i=0;i<5;i++)
    {
        ingineer[i].putdata();
    }
    getch();
    return(0);
}

```

REPLACE AND SORT USING CLASS:-

```
#include<iostream.h>
#include<constream.h>

class sort
{
private:
    int nm[30];
public:
    void getdata();
    void putdata();
};

void sort :: getdata()
{
    int i,j,k;
    cout<<"enter 10 nos:" ;
    for(i=0;i<10;i++)
    {
        cin>>nm[i];
    }
    for(i=0;i<9;i++)
    {
        for(j=i+1;j<10;j++)
        {
            if(nm[i]>nm[j])
            {
                k=nm[i];
                nm[i]=nm[j];
                nm[j]=k;
            }
        }
    }

    void sort :: putdata()
    {
        int k;
        for(k=0;k<10;k++)
        {
            cout<<num [k] <<endl ;
        }
    }
}
```



```

    }
    }
    int main()
    {
        clrscr();
        sort s;
        s.getdata();
        s.putdata();
        return(0);
    }

```

ARRAY OF MEMBERS:

```

#include<iostream.h>
#include<constream.h>

const int m=50;

class items
{
    int item_code[m];
    float item_price[m];
    int count;

public:
    void cnt(void) { count=0;}
    void get_item(void);
    void display_sum(void);
    void remove(void);
    void display _item(void);
};

```

```

void items :: get_item (void)
{
    cout<<"enter itemcode:";
    cin>> item_code[code];
    cout<<"enter item cost:";
    cin>>item_price[count];
    count ++ ;
}

```

```

void items :: display _sum(void)
{
    float sum=0;
    for( int i=0;i<count;i++)
    {

```

```

        sum=sum+item_price[i];
    }
    cout<< "\n total value:"<<sum<<endl;
}
int main ( )
{
    items order;
    order.cnt();
    int x;
    do
    {
        cout<<"\nyou can do the following:";
        cout<<"enter appropriate no:";
        cout<<endl<<" 1 :add an item";
        cout<<endl<<"2: display total value :";
        cout<<endl<<"3 : display an item";
        cout<<endl<<"4 :display all item:";
        cout<<endl<<"5 : quit:";
        cout<<endl<<endl<<"what is your option:";
        cin>>x;

        switch(x)
        {
            case 1: order.get_item(); break;
            case 2: order.display_sum(); break;
            case 3: order.remove(); break;
            case 4: order.display_item();break;
            case 5: break;
            default : cout<<"error in input; try again";
        }
    } while(x!=5);
}

```

LECTURE-17

STATIC DATA MEMBER:

A data member of a class can be qualified as static . The properties of a static member variable are similar to that of a static variable. A static member variable has contain special characteristics.

Variable has contain special characteristics:-

- 1) It is initialized to zero when the first object of its class is created.No other initialization is permitted.
- 2) Only one copy of that member is created for the entire class and is shared by all the objects of that class, no matter how many objects are created.
- 3) It is visible only with in the class but its life time is the entire program. Static variables are normally used to maintain values common to the entire class. For example a static data member can be used as a counter that records the occurrence of all the objects.

```
int item :: count; // definition of static data member
```

Note that the type and scope of each static member variable must be defined outside the class definition .This is necessary because the static data members are stored separately rather than as a part of an object.

Example :-

```
#include<iostream.h>
class item
{
    static int count; //count is static
    int number;
public:
    void getdata(int a)
    {
        number=a;
        count++;
    }
    void getcount(void)
    {
        cout<<"count:";
        cout<<count<<endl;
    }
};

int item :: count ; //count defined
int main( )
{
    item a,b,c;
    a.get_count( );
    b.get_count( );
    c.get_count( );
    a.getdata( );
    b.getdata( );
```

```

        c.getdata( );
        cout<<"after reading data : "<<endl;
        a.get_count( );
        b.get_count( );
        c.get count( );

    return(0);
}

```

The output would be

```

count:0
count:0
count:0

```

After reading data

```

count: 3
count:3
count:3

```

The static Variable count is initialized to Zero when the objects created . The count is incremented whenever the data is read into an object. Since the data is read into objects three times the variable count is incremented three times. Because there is only one copy of count shared by all the three object, all the three output statements cause the value 3 to be displayed.

STATIC MEMBER FUNCTIONS:-

A member function that is declared static has following properties :-

1. A static function can have access to only other static members declared in the same class.
2. A static member function can be called using the class name as follows:-
class - name :: function - name;

Example:-

```

#include<iostream.h>
class test
{
    int code;
    static int count; // static member variable
public:
    void set(void)
    {
        code=++count;
    }
    void showcode(void)
    {
        cout<<"object member : "<<code<<endl;
    }
    static void showcount(void)
    { cout<<"count="<<count<<endl; }
};

int test:: count;
int main()
{

```

```

test t1,t2;
t1.setcode( );
t2.setcode( );
test :: showcount ( );
test t3;
t3.setcode( );
test:: showcount( );
t1.showcode( );
t2.showcode( );
t3.showcode( );
return(0);

```

output:- count : 2
count: 3
object number 1
object number 2
object number 3

OBJECTS AS FUNCTION ARGUMENTS

Like any other data type, an object may be used as A function argument. This can come in two ways

1. A copy of the entire object is passed to the function.
2. Only the address of the object is transferred to the function

The first method is called pass-by-value. Since a copy of the object is passed to the function, any change made to the object inside the function do not effect the object used to call the function.

The second method is called pass-by-reference . When an address of the object is passed, the called function works directly on the actual object used in the call. This means that any changes made to the object inside the functions will reflect in the actual object .The pass by reference method is more efficient since it requires to pass only the address of the object and not the entire object.

Example:-

```

#include<iostream.h>
class time
{
    int hours;
    int minutes;
public:
    void gettime(int h, int m)
    {
        hours=h;
        minutes=m;
    }
    void puttime(void)
    {
        cout<< hours<<"hours and:";

        cout<<minutes<<"minutes:"<<end;
    }
}

```

```

        void sum( time ,time);
    };
void time :: sum (time t1,time t2)    .
{
    minutes=t1.minutes + t2.minutes;
    hours=minutes% 60;
    minutes=minutes% 60;
    hours=hours+t 1.hours+t2.hours;
}

int main()
{
    time T1,T2,T3;
    T1.gettime(2,45);
    T2.gettime(3,30);
    T3.sum(T1,T2);
    cout<<"T1=";
    T1.puttime( );
    cout<<"T2=";
    T2.puttime( );
    cout<<"T3=";
    T3.puttime( );
    return(0);
}

```

LECTURE-18

FRIENDLY FUNCTIONS:-

We know private members can not be accessed from outside the class. That is a non - member function can't have an access to the private data of a class. However there could be a case where two classes manager and scientist, have been defined we should like to use a function income-tax to operate on the objects of both these classes.

In such situations, c++ allows the common function to be made friendly with both the classes , there by following the function to have access to the private data of these classes .Such a function need not be a member of any of these classes.

To make an outside function "friendly" to a class, we have to simply declare this function as a friend of the classes as shown below :

```
class ABC
{
-----
-----
public:
-----
-----
    friend void xyz(void);
};
```

The function declaration should be preceded by the keyword friend , The function is defined else where in the program like a normal C ++ function . The function definition does not use their the keyword friend or the scope operator :: . The functions that are declared with the keyword friend are known as friend functions. A function can be declared as a friend in any no of classes. A friend function, as though not a member function , has full access rights to the private members of the class.

A friend function processes certain special characteristics:

- It is not in the scope of the class to which it has been declared as friend.
- Since it is not in the scope of the class, it cannot be called using the object of that class. It can be invoked like a member function without the help of any object.
- Unlike member functions.

Example:

```
#include<iostream.h>
class sample
{
    int a;
    int b;
public:
    void setvalue( ) { a=25;b=40;}
    friend float mean( sample s);
}
float mean (sample s)
{
    return (float(s.a+s.b)/2.0);
}

int main ( )
{
```

```

sample x;
x . setvalue( );
cout<<"mean value="<<mean(x)<<endl;
return(0);

}

```

output:
mean value : 32.5

A function friendly to two classes

```

#include<iostream.h>
class xyz;
class xyz
{
    int x;
public:
    void setvalue(int x) { x-= I; }
    friend void max (xyz,abc);
};
class abc
{
    int a;
public:
    void setvalue( int i) {a=i; }
    friend void max(xyz,abc);
};

void max( xyz m, abc n)
{
    if(m . x >= n.a)
        cout<<m.x;
    else
        cout<< n.a;
}

int main( )
{
    abc j;
    j . setvalue( 10);
    xyz s;
    s.setvalue(20);
    max( s , j );
    return(0);
}

```

SWAPPING PRIVATE DATA OF CLASSES:

```

#include<iostream.h>

class class-2;
class class-1
{

```



```

        int value 1;
public:
    void indata( int a) { value=a; }
    void display(void) { cout<<value<<endl; }
    friend void exchange ( class-1 &, class-2 &);
};

class class-2
{
    int value2;
public:
    void indata( int a) { value2=a; }
    void display(void) { cout<<value2<<endl; }
    friend void exchange(class-1 & , class-2 &);
};

void exchange ( class-1 &x, class-2 &y)
{
    int temp=x. value 1;
    x. value I=y.valuo2;
    y.value2=temp;
}

int main( )
{
    class-1 c1;
    class-2 c2;
    c1.indata(100);
    c2.indata(200);
    cout<<"values before exchange:"<<endl;
    c1.display( );
    c2.display( );
    exchange (c1,c2);
    cout<<"values after exchange :"<< endl;
    c1. display ( );
    c2. display ( );
    return(0);
}

```

output:

```

values before exchange
    100
    200
values after exchange
    200
    100

```

PROGRAM FOR ILLUSTRATING THE USE OF FRIEND FUNCTION:

```
#include< iostream.h>
class account1;
class account2
{
private:
    int balance;
public:
    account2( ) { balance=567; }
    void showacc2( )
    {
        cout<<"balanceinaccount2 is:"<<balance<<endl;
friend int transfer (account2 &acc2, account1 &acc1,int amount);
    };
class account1
{
private:
    int balance;
public:
    account1 ( ) { balance=345; }

    void showacc1 ( )
    {
        cout<<"balance in account1 :"<<balance<<endl;
    }
friend int transfer (account2 &acc2, account1 &acc1 ,int amount);
};

int transfer ( account2 &acc2, account1 & acc1, int amount)
{
    if(amount <=acc1 . bvalance)
    {
        acc2. balance + = amount;
        acc1 .balance - = amount;
    }
    else
        return(0);
}

int main()
{
    account1  aa;
    account2  bb;

    cout << "balance in the accounts before transfer:" ;
    aa . showacc1( );
    bb . showacc2( );
    cout << "amt transferred from account1 to account2 is:";
    cout<<transfer ( bb,aa,100)<<endl;
```

```
        cout<< " balance in the accounts after the transfer:";
        aa . showacc 1 ( );
        bb. showacc 2( );
        return(0);
}
```

output:

balance in the accounts before transfer

balance in account 1 is 345

balance in account2 is 567

and transferred from account1 to account2 is 100

balance in account 1 is 245

balance in account2 is 667

LECTURE-19

RETURNING OBJECTS:

```
# include< iostream,h>
class complex
{
    float x;
    float y;
public:
    void input( float real , float imag)
    {
        x=real;
        y=imag;
    }
    friend complex sum( complex , complex);
    void show ( complex );
};

complex sum ( complex c1, complex c2)
{
    complex c3;
    c3.x=c1.x+c2.x;
    c3.y=c1.y+c2.y;
    return c3;}

void complex :: show ( complex c)
{
    cout<<c.x<<" +j "<<c.y<<endl;
}
```

```
int main( )
{
    complex a, b,c;
    a.input(3.1,5.65);
    b.input(2.75,1.2);
    c=sum(a,b);
    cout <<" a="; a.show(a);
    cout <<" b= "; b.show(b);
    cout <<" c=" ; c.show(c);
    return(0);
}
```

output:

```
a =3.1 + j 5.65
b= 2.75+ j 1.2
c= 5.55 + j 6.85
```

POINTER TO MEMBERS:

It is possible to take the address of a member of a class and assign it to a pointer. The address of a member can be obtained by applying the operator & to a “fully qualified” class member name.

A class member pointer can be declared using the operator :: * with the class name.

For Example:

```
class A
{
private:
    int m;
public:
    void show( );
};
```

We can define a pointer to the member m as follows :

```
int A :: * ip = & A :: m
```

The ip pointer created thus acts like a class member in that it must be invoked with a class object. In the above statement. The phrase A :: * means “pointer - to - member of a class” . The phrase & A :: m means the “ Address of the m member of a class”

The following statement is not valid :

```
int *ip=&m ; // invalid
```

This is because m is not simply an int type data. It has meaning only when it is associated with the class to which it belongs. The scope operator must be applied to both the pointer and the member.

The pointer ip can now be used to access the m inside the member function (or friend function).

Let us assume that “a” is an object of “ A” declared in a member function . We can access "m" using the pointer ip as follows.

```
cout<< a . * ip;
cout<< a.m;
ap=&a;
cout<< ap-> * ip;
cout<<ap->a;
```

The deferencing operator ->* is used as to accept a member when we use pointers to both the object and the member. The dereferencing operator. .* is used when the object itself is used with the member pointer. Note that * ip is used like a member name.

We can also design pointers to member functions which ,then can be invoked using the deferencing operator in the main as shown below.

```
(object-name.* pointer-to-member function)
(pointer-to -object -> * pointer-to-member function)
```

The precedence of () is higher than that of .* and ->* , so the parenthesis are necessary.

DEREFERENCING OPERATOR:

```
#include<iostream.h>
class M
{
    int x;
    int y;
public:
    void set_xy(int a,int b)
    {
        x=a;
        y=b;
    }
friend int sum(M);
};

int sum (M m)
{
    int M :: * px= &M :: x; //pointer to member x

    int M :: * py= & m ::y;//pointer to y
    M * pm=&m;
    int s=m.* px + pm->py;
    return(s);
}

int main ( )
{
    M m;
    void(M::*pf)(int,int)=&M::set_xy;//pointer to function set-xy (n*pf)( 10,20);
    //invokes set-xy
    cout<<"sum="<<sum(n)<<endl;
    n *op=&n; //point to object n
    ( op->* pf)(30,40); // invokes set-xy
    cout<<"sum="<<sum(n)<<endl ;
    return(0);
}

output:
sum= 30
sum=70
```

LECTURE-20

CONSTRUCTOR:

A constructor is a special member function whose task is to initialize the objects of its class . It is special because its name is the same as the class name. The constructor is invoked when ever an object of its associated class is created. It is called constructor because it construct the values of data members of the class.

A constructor is declared and defined as follows:

```
//class with a constructor
class integer
{
    int m,n;
public:
    integer! void);//constructor declared
    -----
    -----
};
integer :: integer(void)
{
    m=0;
    n=0;
}
```

When a class contains a constructor like the one defined above it is guaranteed that an object created by the class will be initialized automatically.

For example:-

```
Integer int1; //object int 1 created
```

This declaration not only creates the object int1 of type integer but also initializes its data members m and n to zero.

A constructor that accept no parameter is called the default constructor. The default constructor for class A is A :: A(). If no such constructor is defined, then the compiler supplies a default constructor .

Therefore a statement such as :-

```
A a ;//invokes the default constructor of the compiler of the
compiler to create the object "a" ;
```

Invokes the default constructor of the compiler to create the object a.

The constructor functions have some characteristics:-

- They should be declared in the public section .
- They are invoked automatically when the objects are created.
- They don't have return types, not even void and therefore they cannot return values.
- They cannot be inherited , though a derived class can call

the base class constructor .

- Like other C++ function , they can have default arguments,
- Constructor can't be virtual.
- An object with a constructor can't be used as a member of union.

Example of default constructor:

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

class abc
{
private:
    char nm[];
public:
    abc ( )
    {
        cout<<"enter your name:";
        cin>>nm;
    }
    void display( )

{
    cout<<nm;
}

};

int main( )
{
    clrscr( );
    abc d;
    d.display( );
    getch( );
    return(0);
}
```

PARAMETERIZED CONSTRUCTOR:-

the constructors that can take arguments are called parameterized constructors. Using parameterized constructor we can initialize the various data elements of different objects with different values when they are created.

Example:-

```
class integer
{
    int m,n;
public:
    integer( int x, int y);
    -----
    -----
};
```



```
integer:: integer (int x, int y)
{
    m=x;n=y;
}
```

the argument can be passed to the constructor by calling the constructor implicitly.

```
integer int 1 = integer(0,100); // explicit call
integer int 1(0,100); //implicite call
```

CLASS WITH CONSTRUCTOR:-

```
#include<iostream.h>
class integer
{
    int m,n;
public:
    integer(int,int);
    void display(void)

{
    cout<<"m="<<m ;
    cout<<"n="<<n;
}

};

integer :: integer( int x,int y) // constructor defined
{
    m=x;
    n=y;
}

int main( )
{
    integer int 1(0, 100); // implicit call
    integer int2=integer(25,75);
    cout<<" \nobject1 "<<<endl;
    int1.display( );
    cout<<" \n object2 "<<<endl;
    int2.display( );
}
```

output:

```
object 1
m=0
n=100
object2
m=25
n=25
```

Example:-

```
#include<iostream.h>
#include<conio.h>
class abc
{
private:
    char nm [30];
    int age;
public:
    abc ( ){ } // default
    abc ( char x[], int y);
    void get( )
    {
        cout<<"enter your name:";
        cin>>nm;
        cout<<" enter your age:";
        cin>>age;
    }
    void display( )
    {
        cout<<nm<<endl;
        cout<<age;
    }
};

abc :: abc(char x[], int y)
{
    strcpy(nm,x);
    age=y;
}

void main( )
{
    abc l;
    abc m=abc("computer",20000);
    l.get();
    l.display( );
    m.display ( );
    getch( );
}
```

OVERLOADED CONSTRUCTOR:-

```
#include<iostream.h>
#include<conio.h>
class sum
{
private:
    int a;
    int b;
    int c;
    float d;
    double e;
public:
    sum ( )
```

```

{
cout<<"enter a;";
cin>>a;
cout<<"enter b;";
cin>>b;
cout<<"sum= "<<a+b<<endl;
}
sum(int a,int b);
sum(int a, float d,double c);
};
sum :: sum(int x,int y)
{
    a=x;
    b=y;
}
sum :: sum(int p, float q ,double r)
{
    a=p;
    d=q;
    e=r;
}
void main( )
{
clrscr( );
sum l;
sum m=sum(20,50);
sum n= sum(3,3.2,4.55);
getch( );
}

```

output:
enter a : 3
enter b : 8
sum=11
sum=70
sum=10.75

COPY CONSTRUCTOR:

A copy constructor is used to declare and initialize an object from another object.

Example:-

the statement
integer 12(11);

would define the object 12 and at the same time initialize it to the values of 11.

Another form of this statement is : integer 12=11;

The process of initialization through a copy constructor is known as copy initialization.

Example:-

```

#include<iostream.h>
class code
{
    int id;

```

```

        public
            code ( ) { } //constructor
            code (int a) { id=a; } //constructor
            code(code &x)
            {
                Id=x.id;
            }
            void display( )
            {
                cout<<id;
            }
    };

int main( )
{
    code A(100);
    code B(A);
    code C=A;
    code D;
    D=A;
    cout<<" \n id of A :"; A.display( );
    cout<<" \nid of B :"; B.display( );
    cout<<" \n id of C:"; C.display( );
    cout<<" \n id of D:"; D.display( );
}

```

output :-

```

id of A:100
id of B:100
id of C:100
id of D:100

```

DYNAMIC CONSTRUCTOR:-

The constructors can also be used to allocate memory while creating objects . This will enable the system to allocate the right amount of memory for each object when the objects are not of the same size, thus resulting in the saving of memory.

Allocate of memory to objects at the time of their construction is known as dynamic constructors of objects. The memory is allocated with the help of new operator.

Example:-

```

#include<iostream.h>
#include<string.h>
class string
{
    char *name;

    int length;
public:
    string ( )

```

```

    {
        length=0;
        name= new char [length+1]; /* one extra for \0 */
    }
string( char *s) //constructor 2
{
    length=strlen(s);
    name=new char [length+1];
    strcpy(name,s);
}
void display(void)
{
    cout<<name<<endl;
}
void join(string &a .string &b)
{
    length=a. length +b . length;
    delete name;
    name=new char[length+1]; /* dynamic allocation */
    strcpy(name,a.name);
    strcat(name,b.name);
}
};
int main( )
{
    char * first = “Joseph” ;
    string name1(first),name2(“louis”),name3( “LaGrange”),s1,s2;
    s1.join(name1,name2);
    s2.join(s1,name3);
    name1.display( );
    name2.display( );
    name3.display( );
    s1.display( );
    s2.display( );
}

```

output :-

```

    Joseph
    Louis
    language
    Joseph Louis
    Joseph Louis Language

```

LECTURE-21

DESTRUCTOR:-

A destructor, as the name implies, is used to destroy the objects that have been created by a constructor. Like a constructor, the destructor is a member function whose name is the same as the class name but is preceded by a tilde.

For Example:-

```
~ integer( ) { }
```

A destructor never takes any argument nor does it return any value. It will be invoked implicitly by the compiler upon exit from the program to clean up storage that is no longer accessible. It is a good practice to declare a destructor in a program since it releases memory space for future use.

Delete is used to free memory which is created by new.

Example:-

```
matrix : : ~ matrix( )
{
    for(int i=0; i<11;i++)
        delete p[i];
    delete p;
}
```

IMPLEMENTED OF DESTRUCTORS:-

```
#include<iostream.h>
int count=0;
class alpha
{
public:
    alpha( )
    {
        count ++;
        cout<<"\n no of object created :"<<endl;
    }
    ~alpha( )
    {
        cout<<"\n no of object destroyed : "<<endl;
        count--;
    }
};
```

```
int main( )
{
    cout<<"\n\n enter main \n:";
    alpha A1,A2,A3,A4;
    {
        cout<<"\n enter block 1 :\n";
```

```

        alpha A5;
    }
    {
        cout<<"\n\n enter block2 \n";
        alpha A6;
    }
    cout<<"\n re-enter main \n:";
    return(0);
}

```

output:-

```

enter main
no of object created 1
no of object created 2
no of object created 3
no of object created 4
enter block 1
no of object created 5
no of object destroyed 5
enter block 2
no of object created 5
no of object destroyed 5
re-enter main
no of object destroyed 4
no of object created 3
no of object created 2
no of object created 1

```

Example :-

```

#include<iostream.h>
int x=1;
class abc
{
public:
    abc( )
    {
        x--;
        cout<<"construct the no"<<x<<endl;
    }
    ~abc( )
    {
        cout<<"destruct the no:"<<x<<endl;
        x--;
    }
};

int main( )
{
    abc I1,I2,I3,I4;
    cout<<I1<<I2<<I3<<I4<<endl;
    return(0);
}

```

LECTURE-22 & 23

OPERATOR OVERLOADING:-

Operator overloading provides a flexible option for the creation of new definitions for most of the C++ operators. We can overload all the C++ operators except the following:

- Class members access operator (.,.*)
- Scope resolution operator (:)
- Size operator(sizeof)
- Condition operator (? :)

Although the semantics of an operator can be extended, we can't change its syntax, the grammatical rules that govern its use such as the no of operands precedence and associativity. For example the multiplication operator will enjoy higher precedence than the addition operator.

When an operator is overloaded, its original meaning is not lost. For example, the operator +, which has been overloaded to add two vectors, can still be used to add two integers.

DEFINING OPERATOR OVERLOADING:

To define an additional task to an operator, we must specify what it means in relation to the class to which the operator is applied. This is done with the help of a special function called operator function, which describes the task.

Syntax:-

```
return-type class-name :: operator op( arg-list)
{
    function body
}
```

Where return type is the type of value returned by the specified operation and op is the operator being overloaded. The op is preceded by the keyword operator, operator op is the function name.

operator functions must be either member function, or friend function. A basic difference between them is that a friend function will have only one argument for unary operators and two for binary operators, This is because the object used to invoke the member function is passed implicitly and therefore is available for the member functions. Arguments may be either by value or by reference.

operator functions are declared in the class using prototypes as follows:-

```
vector operator + (vector); // vector addition
vector operator - (); //unary minus
friend vector operator + (vector, vector); // vector add
friend vector operator -(vector); // unary minus
vector operator - ( vector &a); // subtraction
int operator ==(vector); //comparision
friend int operator ==(vector, vector); // comparision
```

vector is a data type of class and may represent both magnitude and direction or a series of points called elements.

The process of overloading involves the following steps:-

1. Create a class that defines the data type that is used in the overloading operation.
2. Declare the operator function operator op() in the public part of the class
3. It may be either a member function or friend function.
4. Define the operator function to implement the required operations.

Overloaded operator functions can be invoked by expressions such as

op x or x op;

for unary operators and

x op y

for binary operators.

operator op(x);

for unary operator using friend function

operator op(x,y);

for binary operator using friend function.

Unary – operator overloading(using member function):

```
class abc
{
int m,n;
public:
abc()
{
m=8;
n=9;
}
void show()
{
cout<<m<<n;
}
operator -- ()
{
--m;
--n;
}
};

void main()
{
abc x;
x.show();
--x;
```

```
x.show();  
}
```

Unary – - operator overloading(using friend function):

```
class abc  
{  
int m,n;  
public:  
abc()  
{  
m=8;  
n=9;  
}  
void show()  
{  
cout<<m<<n;  
}  
friend operator --(abc &p);  
};  
operator -- (abc &p)  
{  
--p.m;  
--p.n;  
}  
};  
void main()  
{  
abc x;  
x.show();  
operator--(x);  
x.show();  
}
```

Unary operator+ for adding two complex numbers (using member function)

```
class complex
{
float real,img;
public:
complex()
{
    real=0;
    img=0;
}
complex(float r,float i)
{
real=r;
img=i;
}
void show()
{
cout<<real<<"+"i"<<img;
}
complex operator+(complex &p)
{
    complex w;
    w.real=real+p.real;
    w.img=img+p.img;
    return w;
}
};
void main()
{
complex s(3,4);
complex t(4,5);
complex m;
m=s+t;
s.show();
t.show();
m.show();
}
```

Unary operator+ for adding two complex numbers (using friend function)

```
class complex
{
float real,img;
public:
complex()
{
    real=0;
    img=0;
}
complex(float r,float i)
{
real=r;
img=i;
```

```

}
void show()
{
cout<<real<<"i"<<img;
}
friend complex operator+(complex &p,complex &q);
};
complex operator+(complex &p,complex &q)
{
    complex w;
    w.real=p.real+q.real;
    w.img=p.img+q.img;
    return w;
}
};
void main()
{
complex s(3,4);complex t(4,5);
complex m;
m=operator+(s,t);
s.show();t.show();
m.show();
}

```

Overloading an operator does not change its basic meaning. For example assume the + operator can be overloaded to subtract two objects. But the code becomes unreachable.

```

class integer
{
    int x, y;
    public:
    int operator + ( ) ;
}
int integer: : operator + ( )
{
    return (x-y) ;
}

```

Unary operators, overloaded by means of a member function, take no explicit argument and return no explicit values. But, those overloaded by means of a friend function take one reference argument (the object of the relevant class).

Binary operators overloaded through a member function take one explicit argument and those which are overloaded through a friend function take two explicit arguments.

Table 7.2

Operator to Overload	Arguments passed to the Member Function	Arguments passed to the Friend Function
Unary Operator	No	1
Binary Operator	1	2

LECTURE-24

Type Conversions

In a mixed expression constants and variables are of different data types. The assignment operations causes automatic type conversion between the operand as per certain rules.

The type of data to the right of an assignment operator is automatically converted to the data type of variable on the left.

Consider the following example:

```
int x;  
float y = 20.123;  
x=y ;
```

This converts float variable y to an integer before its value assigned to x. The type conversion is automatic as far as data types involved are built in types. We can also use the assignment operator in case of objects to copy values of all data members of right hand object to the object on left hand. The objects in this case are of same data type. But of objects are of different data types we must apply conversion rules for assignment.

There are three types of situations that arise where data conversion are between incompatible types.

1. Conversion from built in type to class type.
2. Conversion from class type to built in type.
3. Conversion from one class type to another.

Basic to Class Type

A constructor was used to build a matrix object from an int type array. Similarly, we used another constructor to build a string type object from a char* type variable. In these examples constructors performed a defacto type conversion from the argument's type to the constructor's class type

Consider the following constructor:

```
string :: string (char*a)  
{  
    length = strlen (a);  
    name=new char[len+1];  
    strcpy (name,a);  
}
```

This constructor builds a string type object from a char* type variable a. The variables length and name are data members of the class string. Once you define the constructor in the class string, it can be used for conversion from char* type to string type.

Example

```
string s1 , s2;  
char* name1 = "Good Morning";  
char* name2 = "STUDENTS";  
s1 = string(name1);  
s2 = name2;
```

The program statement

```
si = string (name1);
```

first converts name 1 from char* type to string type and then assigns the string type values to the object s1. The statement

```
s2 = name2;
```

performs the same job by invoking the constructor implicitly.

Consider the following example

```
class time
{
    int hours;
    int minutes;
    public:
    time (int t) // constructor
    {
        hours = t / 60;           //t is inputted in minutes
        minutes = t % 60;
    }
};
```

In the following conversion statements :

```
time T1;           //object T1 created
int period = 160;
T1 = period;       //int to class type
```

The object T1 is created. The variable period of data type integer is converted into class type time by invoking the constructor. After this conversion, the data member hours of T1 will have value 2 and minutes will have a value of 40 denoting 2 hours and 40 minutes.

Note that the constructors used for the type conversion take a single argument whose type is to be converted.

In both the examples, the left-hand operand of = operator is always a class object. Hence, we can also accomplish this conversion using an overloaded = operator.

LECTURE-25

Class to Basic Type

The constructor functions do not support conversion from a class to basic type. C++ allows us to define a overloaded casting operator that convert a class type data to basic type. The general form of an overloaded casting operator function, also referred to as a conversion function, is:

```
operator typename ( )
{
    //Program statmerit .
}
```

This function converts a class type data to typename. For example, the operator double() converts a class object to type double, in the following conversion function:

```
vector:: operator double ( )
{
    double sum = 0 ;
    for(int I = 0; ioize;
    sum = sum + v[i] * v[i ] ; //scalar magnitude
    return sqrt(sum);
}
```

The casting operator should satisfy the following conditions.

- It must be a class member.
- It must not specify a return type.
- It must not have any arguments. Since it is a member function, it is invoked by the object and therefore, the values used for, Conversion inside the function belongs to the object that invoked the function. As a result function does not need an argument.

In the string example discussed earlier, we can convert the object string to char* as follows:

```
string:: operator char*( )
{
    return (str) ;
}
```

One Class to Another Class Type

We have just seen data conversion techniques from a basic to class type and a class to basic type. But sometimes we would like to convert one class data type to another class type.

Example

Obj1 = Obj2 ; //Obj1 and Obj2 are objects of different classes.

Obj1 is an object of class one and Obj2 is an object of class two. The class two type data is converted to class one type data and the converted value is assigned to the Obj1. Since the conversion takes place from class two to class one, two is known as the source and one is known as the destination class.

Such conversion between objects of different classes can be carried out by either a constructor or a conversion function. Which form to use, depends upon where we want the type-conversion function to be located, whether in the source class or in the destination class.

We studied that the casting operator function

Operator typename()

Converts the class object of which it is a member to typename. The type name may be a built-in type or a user defined one(another class type) . In the case of conversions between objects,

typename refers to the destination class. Therefore, when a class needs to be converted, a casting operator function can be used. The conversion takes place in the source class and the result is given to the destination class object.

Let us consider a single-argument constructor function which serves as an instruction for converting the argument's type to the class type of which it is a member. The argument belongs to the source class and is passed to the destination class for conversion. Therefore the conversion constructor must be placed in the destination class.

Table 7.3

Conversion	Conversion takes place in	
	Source class	Destination class
Basic to class	Not applicable	Constructor
Class to Basic	Casting operator	Not applicable
Class to class	Casting operator	Constructor

When a conversion using a constructor is performed in the destination class, we must be able to access the data members of the object sent (by the source class) as an argument. Since data members of the source class are private, we must use special access functions in the source class to facilitate its data flow to the destination class.

Consider the following example of an inventory of products in a store. One way of keeping record of the details of the products is to record their code number, total items in the stock and the cost of each item. Alternatively we could just specify the item code and the value of the item in the stock. The following program uses classes and shows how to convert data of one type to another.

```
#include<iostream.h>
#include<conio.h>
class stock2;
class stock1
{
int code, item;
float price;
public:
stock1 (int a, int b, float c)
{
code=a;
item=b;
price=c;
}
void disp( )
{
cout<<"code"<<code <<"\n";
cout<<"Items"<<item <<"\n";
cout<<"Price per item Rs . "<<price <<"\n";
}
int getcode( )
{return code; }
int getitem( )
{return item; }
int getprice( )
{return price;}
```



```

operator float( )
{
return ( item*price );
}
};

class stock2
{
int code;
float val;
public:
stock2()
{
code=0; val=0;
}
stock2(int x, float y)
{
code=x; val=y;
}
void disp( )
{
cout<< "code"<<code << "\n";
cout<< "Total Value Rs . " <<val <<"\n"
}
stock2 (stock1 p)
{
code=p . getcode ( ) ;
val=p.getitem( ) * p. getprice ( ) ;
}
};

void main ( )
{
Stock1 i1(101, 10,125.0);
Stock2 i2;
float tot_val;
tot_val=i1 ;
i2=i1 ;
cout<<" Stock Details-stock1-type" <<"\n";
i1 . disp ( ) ;
cout<<" Stock value"<<"\n";
cout<< tot_val<<"\n";
cout<<" Stock Details-stock2-type"<< "\n";
i2 .disp( ) ;
getch ( ) ;
}

```

You should get the following output.

Stock Details-stock1-type

code 101

Items 10

Price per item Rs. 125

Stock value

1250

Stock Details-stock2-type

code 10 1

Total Value Rs. 1250

LECTURE-26

Inheritance:

Reaccessability is yet another feature of OOP's. C++ strongly supports the concept of reusability. The C++ classes can be used again in several ways. Once a class has been written and tested, it can be adopted by another programmers. This is basically created by defining the new classes, reusing the properties of existing ones. The mechanism of deriving a new class from an old one is called 'INHERTTENCE'. This is often referred to as IS-A' relationship because very object of the class being defined "is" also an object of inherited class. The old class is called 'BASE' class and the new one is called'DERIEVED'class.

Defining Derived Classes

A derived class is specified by defining its relationship with the base class in addition to its own details. The general syntax of defining a derived class is as follows:

```
class d_classname : Access specifier baseclass name
{
    —
    — // members of derived class
};
```

The colon indicates that the a-class name is derived from the base class name. The access specifier or the visibility mode is optional and, if present, may be public, private or protected. By default it is private. Visibility mode describes the status of derived features e.g.

```
class xyz      //base class
{
    members of xyz
};
class ABC : public xyz  //public derivation
{
    members of ABC
};
class ABC : XYZ  //private derivation (by default)
{
    members of ABC
};
```

In the inheritance, some of the base class data elements and member functions are inherited into the derived class. We can add our own data and member functions and thus extend the functionality of the base class. Inheritance, when used to modify and extend the capabilities of the existing classes, becomes a very powerful tool for incremental program development.

Single Inheritance

When a class inherits from a single base class, it is known as single inheritance. Following program shows the single inheritance using public derivation.

```
#include<iostream.h>
#include<conio.h>
class worker
{
```

```

    int age;
    char name [10];
    public:
    void get ( );
};
void worker : : get ( )
{
    cout <<"yout name please"
    cin >> name;
    cout <<"your age please" ;
    cin >> age;
}
void worker  :: show ( )
{
    cout <<"In My name is :"<<name<<"In My age is :"<<age;
}
class manager :: public worker  //derived class (publicly)
{
    int now;
    public:
    void get ( ) ;
    void show ( ) ;
};
void manager : : get ( )
{
    worker : : get ( ) ; //the calling of base class input fn.
    cout << "number of workers under you";
    cin >> now;
    cin>>name>>age;
}
// ( if they were public )
void manager :: show ( )
{
    worker :: show ( ) ; //calling of base class o/p fn.
    cout <<"in No. of workers under me are: " << now;
}

main ( )
{
    clrscr ( ) ;
    worker W1;
    manager M1;
    M1 .get ( ) ;
    M1.show ( ) ;
}

```

If you input the following to this program:

Your name please

Ravinder

Your age please

27

number of workers under you

30

Then the output will be as follows:

My name is : Ravinder

My age is : 27

No. of workers under me are : 30

The following program shows the single inheritance by private derivation.

```
#include<iostream.h>
#include<conio.h>
class worker //Base class declaration
{
    int age;
    char name [10] ;
    public:
    void get ( ) ;
    void show ( ) ;
};
void worker : : get ( )
{
    cout << "your name please" ;
    cin >> name;
    cout << "your age please";
    cin >>age;
}
void worker : show ( )
{
    cout << "in my name is: " <<name<< "in" << "my age is : " <<age;
}
class manager : worker //Derived class (privately by default)
{
    int now;
    public:
    void get ( ) ;
    void show ( ) ;
};
void manager : : get ( )
{
    worker : : get ( ) ; //calling the get function of base
    cout << "number of worker under you"; class which is
    cin >> now;
}
void manager : : show ( )
{
    worker : : show ( ) ;
    cout << "in no. of worker under me are : " <<now;
}
main ( )
{
```

```

        clrscr ( ) ;
        worker wl ;
        manager ml;
        ml.get ( ) ;
        ml.show ( );
    }

```

The following program shows the single inheritance using protected derivation

```

#include<conio.h>
#include<iostream.h>
class worker          //Base class declaration
{ protected:
    int age; char name [20];
    public:
        void get ( );
        void show ( );
};
void worker :: get ( )
{
    cout >> "your name please";
    cin >> name;
    cout << "your age please";
    cin >> age;
}
void worker :: show ( )
{
    cout << "in my name is: " << name << "in my age is " << age;
}
class manager:: protected worker // protected inheritance
{
    int now;
    public:
        void get ( );
        void show ( ) ;
};
void manager : : get ( )
{
    cout << "please enter the name In";
    cin >> name;
    cout<< "please enter the age In"; //Directly inputting the data
    cin >> age;    members of base class
    cout << " please enter the no. of workers under you:";
    cin >> now;
}
void manager : : show ( )

{
    cout << "your name is : "<<name<<" and age is : "<<age;
    cout <<"In no. of workers under your are : "<<now;
main ( )
{
    clrscr ( ) ;
    manager ml;
    ml.get ( ) ;
}

```

```

        cout << "\n\n";
        ml.show ();
    }

```

Making a Private Member Inheritable

Basically we have visibility modes to specify that in which mode you are deriving the another class from the already existing base class. They are:

- Private:** when a base class is privately inherited by a derived class, 'public members' of the base class become private members of the derived class and therefore the public members of the base class can be accessed by its own objects using the dot operator. The result is that we have no member of base class that is accessible to the objects of the derived class.
- Public:** On the other hand, when the base class is publicly inherited, 'public members' of the base class become 'public members' of derived class and therefore they are accessible to the objects of the derived class.
- Protected:** C++ provides a third visibility modifier, protected, which serve a little purpose in the inheritance. A member declared as protected is accessible by the member functions within its class and any class immediately derived from it. It cannot be accessed by functions outside these two classes.

The below mentioned table summarizes how the visibility of members undergo modifications when they are inherited

Base Class Visibility	Derived Class Visibility		
	Public	Private	Protected
Private	X	X	X
Public	Public	Private	Protected
Protected	Protected	Private	Protected

The private and protected members of a class can be accessed by:

- A function i.e. friend of a class.
- A member function of a class that is the friend of the class.
- A member function of a derived class.

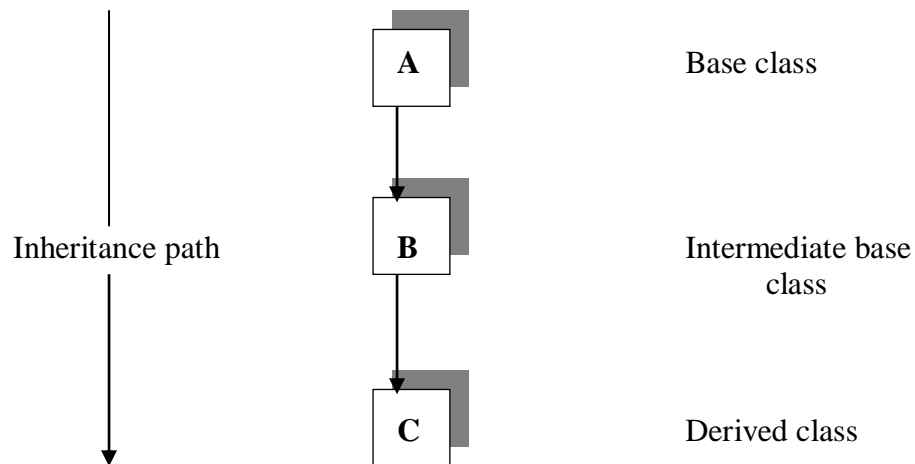
Student Activity

- Define Inheritance. What is the inheritance mechanism in C++?
- What are the advantage of Inheritance?
- What should be the structure of a class when it has to be a base for other classes?

LECTURE-27

Multilevel Inheritance

When the inheritance is such that, the class A serves as a base class for a derived class B which in turn serves as a base class for the derived class C. This type of inheritance is called 'MULTILEVEL INHERITENCE'. The class B is known as the 'INTERMEDIATE BASE CLASS' since it provides a link for the inheritance between A and C. The chain ABC is called 'ITNHERITENCE*PATH' for e.g.



The declaration for the same would be:

```
Class A
{
//body
}
Class B : public A
{
//body
}
Class C : public B
{
//body
}
```

This declaration will form the different levels of inheritance.

Following program exhibits the multilevel inheritance.

```
#include<iostream.h>
#include<conio.h>
class worker    // Base class declaration
{
    int age;
    char name [20] ;
    public;
    void get( ) ;
```



```

        void show( ) ;
    }

void worker: get ( )
{
    cout << "your name please" ;
    cin >> name;
    cout << "your age please" ;
}

void worker : : show ( )
{
    cout << "In my name is : " << name << " In my age is : " << age;
}

class manager : public worker //Intermediate base class derived
{
    //publicly from the base class
    int now;
    public:
    void get ( ) ;
    void show( ) ;
};

void manager :: get ( )
{
    worker : : get ( ) ;    //calling get ( ) fn. of base class
    cout << "no. of workers under you:";
    cin >> now;
}

void manager : : show ( )
{
    worker : : show ( ) ;    //calling show ( ) fn. of base class
    cout << "In no. of workers under me are: " << now;
}

class ceo: public manager    //declaration of derived class
{
    //publicly inherited from the
    int nom;    //intermediate base class
    public:
    void get ( ) ;
    void show ( ) ;
};

void ceo : : get ( )
{
    manager : : get ( ) ;
    cout << "no. of managers under you are:"; cin >> nom;
}

void manager : : show ( )
{
    cout << "In the no. of managers under me are: In";
    cout << "nom;
}

```

```

main ( )
{
    clrscr ( ) ;
    ceo cl ;
    cl.get ( ) ; cout << “\n\n”;
    cl.show ( ) ;
}

```

Worker

Private: int age; char name[20];
Protected:
Private: int age; char name[20];

Manager:Worker

Private: int now;
Protected:
Public: void get() void show() worker ::get() worker ::get()

Ceo: Manager

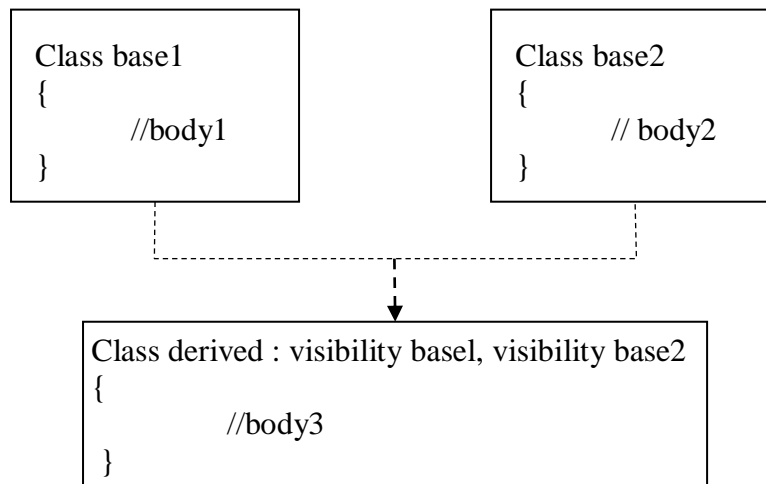
Public:
Protected:
Public:

All the inherited
members

Multiple Inheritances

A class can inherit the attributes of two or more classes. This mechanism is known as ‘MULTIPLE INHERITENCE’. Multiple inheritance allows us to combine the features

of several existing classes as a starting point for defining new classes. It is like the child inheriting the physical feature of one parent and the intelligence of another. The syntax of the derived class is as follows:



Where the visibility refers to the access specifiers i.e. public, private or protected. Following program shows the multiple inheritance.

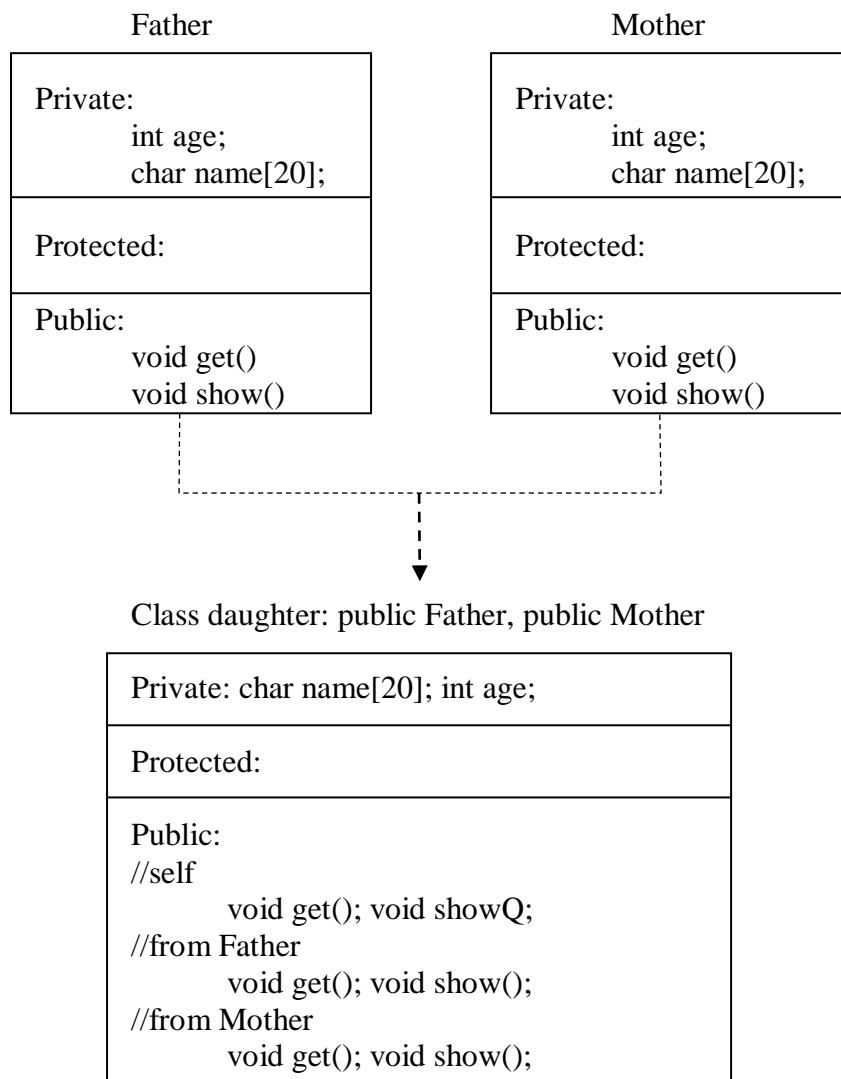
```
#include<iostream.h>
#include<conio . h>
class father          //Declaration of base class1
{
    int age ;
    char flame  [20] ;
    public:
    void get ( ) ;
    void show ( ) ;
};
void father : : get ( )
{
    cout << "your father name please";
    cin >> name;
    cout << "Enter the age";
    cin >> age;
}
void father : : show ( )
{
    cout<< "In my father's name is: ' <<name<< "In my father's age is:<<age;
}
class mother          //Declaration of base class 2
{
    char name [20] ;
    int age ;
```

```

public:
void get ( )
{
    cout << "mother's name please" << "In";
    cin >> name;
    cout << "mother's age please" << "in";
    cin >> age;
}
void show ( )
{
    cout << "In my mother's name is: " << name;
    cout << "In my mother's age is: " << age;
}
class daughter : public father, public mother //derived class inheriting
{
    //publicly
    char name [20] ; //the features of both the base class
    int std;
public:
    void get ( ) ;
    void show ( ) ;
};
void daughter :: get ( )
{
    father :: get ( ) ;
    mother :: get ( ) ;
    cout << "child's name: ";
    cin >> name;
    cout << "child's standard";
    cin >> std;
}
void daughter :: show ( )
{
    father :: show ( ) ;
    mother :: show ( ) ;
    cout << "In child's name is : " << name;
    cout << "In child's standard: " << std;
}
main ( )
{
    clrscr ( ) ;
    daughter d1;
    d1.get ( ) ;
    d1.show ( ) ;
}

```

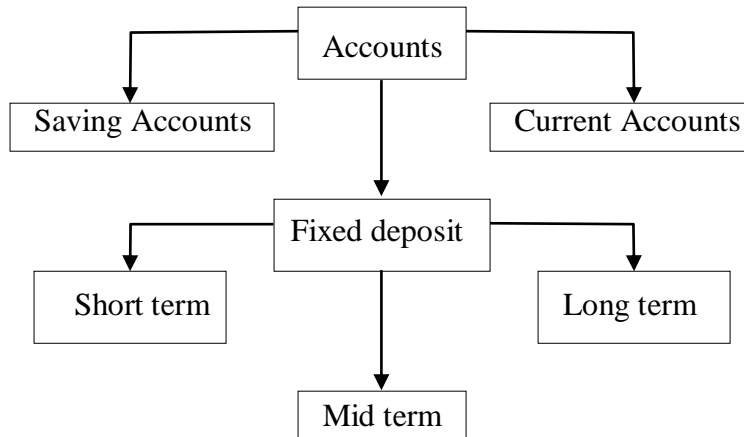
Diagrammatic Representation of Multiple Inheritance is as follows:



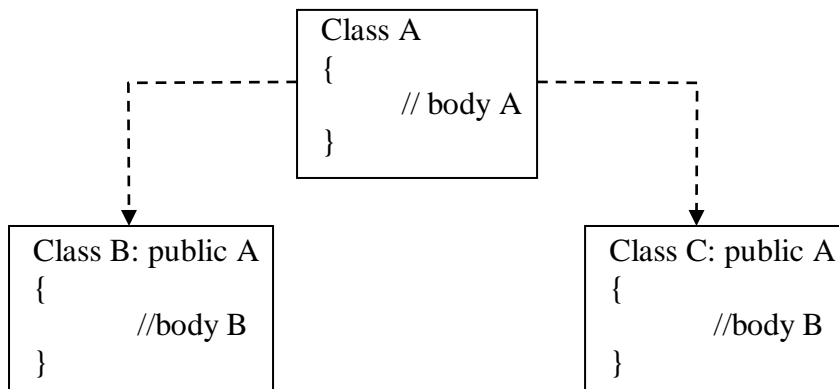
LECTURE-28

Hierarchical Inheritance

Another interesting application of inheritance is to use it as a support to a hierarchical design of a class program. Many programming problems can be cast into a hierarchy where certain features of one level are shared by many others below that level for e.g.



In general the syntax is given as



In C++, such problems can be easily converted into hierarchies. The base class will include all the features that are common to the subclasses. A sub-class can be constructed by inheriting the features of base class and so on.

```
// Program to show the hierarchical inheritance
#include<iostream.h>
#include<conio.h>
class father          //Base class declaration
{
    int age;
    char name [15];
public:
    void get ( )
    {
        cout<< "father name please"; cin >> name;
```

```

        cout<< "father's age please"; cin >> age;
    }
    void show ( )
    {
        cout << "In father's name is ": "<<name;
        cout << "In father's age is: "<< age;
    }
};
class son : public father      //derived class 1
{
    char name [20] ;
    int age ;
    public;
    void get ( ) ;
    void show ( ) ;
};
void son :: get ( )

{
    father :: get ( ) ;
    cout << "your (son) name please" << "in"; cin >>name;
    cout << "your age please" << "In"; cin>>age;
}
void son :: show ( )
{
    father :: show ( ) ;
    cout << "In my name is : " <<name;
    cout << "In my age is : " <<age;
}
class daughter : public father      //derived class 2.
{
    char name [15] ;
    int age;
    public:
    void get ( )
    {
        father :: get ( ) ;
        cout << "your (daughter's) name please In" cin>>name;
        cout << "your age please In"; cin >>age;
    }
    void show ( )
    {
        father :: show ( ) ;
        cout << "in my father name is: " << name << "
        In and his age is : "<<age;
    }
};
main ( )
{
    clrscr ( ) ;

```

```

        son S1;
        daughter D1 ;
        S1. get ( ) ;
        D1. get ( ) ;
        S1 .show( ) ;
        D1. show ( ) ;
    }

```

Hybrid Inheritance

There could be situations where we need to apply two or more types of inheritance to design a program. Basically Hybrid Inheritance is the combination of one or more types of the inheritance. Here is one implementation of hybrid inheritance.

//Program to show the simple hybrid inheritance

```

#include<iostream.h>
#include<conio.h>
class student          //base class declaration
{
    protected:
    int r_no;
    public:
        void get_n (int a)
        {
            r_no =a;
        }
        void put_n (void)
        {
            cout << "Roll No. : "<< r_no;
            cout << "In";
        }
};
class test : public student
{          //Intermediate base class
    protected : int parti, par 2;

    public :
        void get_m (int x, int y) {
            parti = x; part 2 = y; }
        void put_m (void) {
            cout << "marks obtained: " << "In"
                << "Part 1 = " << parti << "in"
                << "Part 2 = " << part2 << "In";
        }
};
class sports          // base for result
{
    protected : int score;
    public:
        void get_s (int s) {
            score = s }
        void put_s (void) {
            cout << " sports wt. : " << score << "\n\n";

```



```

    }
};
class result : public test, public sports //Derived from test
    & sports
{
    int total;
public:
    void display (void);
};

void result::display (void)
{
    total = part1 + part2 + score;
    put_n ( ) ;.
    put_m ( ) ;
    put_S ( ) ;
    cout << "Total score: " << total << "\n"
}
main ( )
{
    clrscr ( ) ;
    result S1;
    S1.get_n (347) ;
    S1.get_m (30, 35);
    S1.get_s (7) ;
    S1.display ( ) ;
}

```

Student Activity

1. What is the major use of multilevel Inheritance?
2. How are arguments sent to the base constructors in multiple inheritance? Whose responsibility is it.
3. What is the difference between hierarchical and hybrid Inheritance.

LECTURE-29

Virtual Base Classes

We have just discussed a situation which would require the use of both multiple and multi level inheritance. Consider a situation, where all the three kinds of inheritance, namely multi-level, multiple and hierarchical are involved.

Let us say the 'child' has two direct base classes 'parent1' and 'parent2' which themselves has a common base class 'grandparent'. The child inherits the traits of 'grandparent' via two separate paths. It can also be inherit directly as shown by the broken line. The grandparent is sometimes referred to as 'INDIRECT BASE CLASS'. Now, the inheritance by the child might cause some problems. All the public and protected members of 'grandparent' are inherited into 'child' twice, first via 'parent1' and again via 'parent2'. So, there occurs a duplicacy which should be avoided.

The duplication of the inherited members can be avoided by making common base class as the virtual base class: for e.g.

```
class g_parent
{
    //Body
};
class parent1: virtual public g_parent
{
    // Body
};

class parent2: public virtual g_parent
{
    // Body
};

class child : public parent1, public parent2
{
    // body
};
```

When a class is virtual base class, C++ takes necessary care to see that only one copy of that class is inherited, regardless of how many inheritance paths exists between virtual base class and derived class. Note that keywords 'virtual' and 'public' can be used in either order.

//Program to show the virtual base class

```
#include<iostream.h>
#include<conio . h>
class student          // Base class declaration
{
    protected:
    int r_no;
    public:
    void get_n (int a)
    { r_no = a; }
    void put_n (void)
    { cout << "Roll No. " << r_no<< "ln";}
};
```

```

class test : virtual public student // Virtually declared common
{
    //base class 1
    protected:
    int part1;
    int part2;
    public:
    void get_m (int x, int y)
    { part1= x; part2=y;}
    void putm (void)
    {
        cout << "marks obtained: " << "\n";
        cout << "part1 = " << part1 << "\n";
        cout << "part2 = " << part2 << "\n";
    }
};

class sports : public virtual student // virtually declared common
{
    //base class 2
    protected:
    int score;
    public:
    void get_s (int a) {
        score = a ;
    }
    void put_s (void)
    { cout << "sports wt.: " << score << "\n"; }
};

class result: public test, public sports //derived class
{
    private : int total ;
    public:
    void show (void) ;
};

void result : : show (void)
{ total = part1 + part2 + score ;
  put_n ( ) ;
  put_m ( ) ;
  put_s ( ) ; cout << "\n total score= " << total << "\n" ;
}

main ( )
{
    clrscr ( ) ;
    result S1 ;
    S1.get_n (345)
    S1.get_m (30, 35) ;
    S1.get-S (7) ;
    S1. show ( ) ;
}

//Program to show hybrid inheritance using virtual base classes
#include<iostream.h>
#include<conio.h>
Class A
{

```

```

protected:
    int x;
public:
    void get (int) ;
    void show (void) ;
};

void A : : get (int a)
    { x = a ; }
void A : : show (void)
    { cout << X ;}
Class A1 : Virtual Public A
{

```

```

protected:
    int y ;
public:
    void get (int) ;
    void show (void);
};

void A1 :: get (int a)
    { y = a;}
void A1 :: show (void)
{
cout <<y ;
{
class A2 : Virtual public A
{
protected:
    int z ;
public:
    void get (int a)
        { z =a;}
    void show (void)
        { cout << z;}
};

class A12 : public A1, public A2
{
int r, t ;
public:
    void get (int a)
        { r = a;}
    void show (void)
        { t = x + y + z + r ;
          cout << "result =" << t ;
        }
};

main ( )
{
clrscr ( ) ;

```

```
A12 r;  
r.A :: get (3) ;  
r.A1 :: get (4) ;  
r.A2 :: get (5) ;  
r.get (6) ;  
r . show ( ) ;  
}
```

LECTURE-30

Polymorphism:

Introduction

When an object is created from its class, the member variables and member functions are allocated memory spaces. The memory spaces have unique addresses. Pointer is a mechanism to access these memory locations using their address rather than the name assigned to them. You will study the implications and applications of this mechanism in detail in this chapter.

Pointer is a variable which can hold the address of a memory location rather than the value at the location. Consider the following statement

```
int num =84;
```

This statement instructs the compiler to reserve a 2-byte of memory location and puts the value 84 in that location. Assume that the compiler allocates memory location 1001 to num. Diagrammatically, the allocation can be shown as:

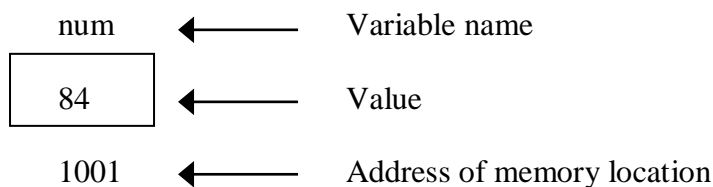


Figure 9.1

As the memory addresses are themselves numbers, they can be assigned to some other variable. For example, ptr be the variable to hold the address of variable num.

Thus, we can access the value of num by the variable ptr. We can say “ptr points to num” as shown in the figure below.

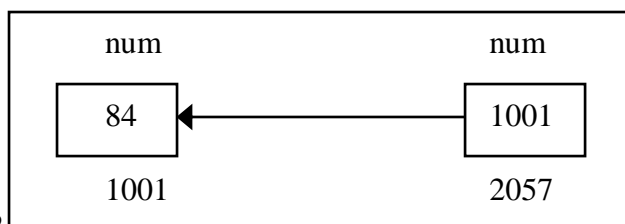


Fig 9.2

Pointers to Objects

An object of a class behaves identically as any other variable. Just as pointers can be defined in case of base C++ variables so also pointers can be defined for an object type. To create a pointer variable for the following class

```
class employee {  
    int code;  
    char name [20] ;  
public:  
    inline void getdata ( )= 0 ;  
    inline void display ( )= 0 ;  
};
```

The following codes is written

```
employee *abc;
```

This declaration creates a pointer variable abc that can point to any object of employee type.

this Pointer

C++ uses a unique keyword called "this" to represent an object that invokes a member function. 'this' is a pointer that points to the object for which this function was called. This unique pointer is called and it passes to the member function automatically. The pointer this acts as an implicit argument to all the member function, for e.g.

```
class ABC  
{  
    int a ;  
    -----  
    -----  
};
```

The private variable 'a' can be used directly inside a member function, like

```
a=123;
```

We can also use the following statement to do the same job.

```
this → a = 123
```

e.g.

```
class stud  
{  
    int a;  
public:  
    void set (int a)  
    {  
        this → a = a; //here this point is used to assign a class level  
    } 'a' with the argument 'a'  
    void show ( )  
    {  
        cout << a;  
    }  
};  
main ( )  
{  
    stud S1, S2;
```

```

        S1.bet (5);
        S2.show ( );
    }
    o/p = 5

```

Pointers to Derived Classes

Polymorphism is also accomplished using pointers in C++. It allows a pointer in a base class to point to either a base class object or to any derived class object. We can have the following Program segment show how we can assign a pointer to point to the object of the derived class.

```

class base
{
    //Data Members
    //Member Functions
};
class derived : public base
{
    //Data Members
    //Member functions
};

void main ( ) {
    base *ptr; //pointer to class base
    derived obj ;
    ptr = &obj ;    //indirect reference obj  to the pointer
    //Other Program statements

}

```

The pointer ptr points to an object of the derived class obj. But, a pointer to a derived class object may not point to a base class object without explicit casting.

For example, the following assignment statements are not valid

```

void main ( )
{
    base obja;
    derived *ptr;
    ptr = &obja; //invalid....explicit casting required
    //Other Program statements
}

```

A derived class pointer cannot point to base class objects. But, it is possible by using explicit casting.

```

void main ( )
{
    base obj ;
    derived *ptr;    // pointer of the derived class
    ptr = (derived *) &obj;    //correct reference
    //Other Program statements
}

```

Student Activity

1. Define Pointers.
2. What are the various operators of pointer? Describe their usage.
3. How will you declare a pointer in C++?

LECTURE-31

Virtual Functions

Virtual functions, one of advanced features of OOP is one that does not really exist but it« appears real in some parts of a program. This section deals with the polymorphic features which are incorporated using the virtual functions.

The general syntax of the virtual function declaration is:

```
class use_detined_name{
private:
public:
virtual return_type function_name1 (arguments);
virtual return_type function_name2(arguments);
virtual return_type function_name3( arguments);
-----
};
```

To make a member function virtual, the keyword virtual is used in the methods while it is declared in the class definition but not in the member function definition. The keyword virtual precedes the return type of the function name. The compiler gets information from the keyword virtual that it is a virtual function and not a conventional function declaration.

For. example, the following declararion of the virtual function is valid.

```
class point {
intx;
inty;
public:
virtual int length ( );
virtual void display ( );
};
```

Remember that the keyword virtual should not be repeated in the definition if the definition occurs outside the class declaration. The use of a function specifier virtual in the function definition is invalid.

For example

```
class point {
intx ;
inty ;
public:
virtual void display ( );
};
virtual void point: : display ( ) //error
{
Function Body
}
```

A virtual function cannot be a static member since a virtual member is always a member of a particular object in a class rather than a member of the class as a whole.

```
class point {
int x ;
int y ;
public:
virtual static int length ( ); //error
```

```
};
    int point: : length ( )
    {
        Function body
    }
```

A virtual function cannot have a constructor member function but it can have the destructor member function.

```
class point {
int x ;
int y ;
public:
virtual point (int xx, int yy) ; // constructors, error
void display ( ) ;
int length ( ) ;
};
```

A destructor member function does not take any argument and no return type can be specified for it not even void.

```
class point {
int x ;
int y ;
public:
virtual point (int xx, int yy) ; //invalid
void display ( ) ;
int length ( ) ;
```

It is an error to redefine a virtual method with a change of return data type in the derived class with the same parameter types as those of a virtual method in the base class.

```
class base {
int x,y ;
public:
virtual int sum (int xx, int yy) ; //error
} ;
class derived: public base {
intz ;
public:
virtual float sum (int xx, int yy) ;
};
```

The above declarations of two virtual functions are invalid. Even though these functions take identical arguments note that the return data types are different.

```
virtual int sum (int xx, int IT) ; //base class
virtual float sum (int xx, int IT) ; //derived class
```

Both the above functions can be written with int data types in the base class as well as in the derived class as

```
virtual int sum (int xx, int yy) ; //base class
virtual int sum (int xx, int yy) ; //derived class
```

Only a member function of a class can be declared as virtual. A non member function (nonmethod) of a class cannot be declared virtual.

```
virtual void display ( ) //error, nonmember function
{
    Function body
}
```

Student Activity

1. What are virtual functions
2. What are pure virtual functions
3. Define Virtual destructors.

Late Binding

As we studied in the earlier unit, late binding means selecting functions during the execution. Though late binding requires some overhead it provides increased power and flexibility. The late binding is implemented through virtual functions as a result we have to declare an object of a class either as a pointer to a class or a reference to a class.

For example the following shows how a late binding or run time binding can be carried out with the help of a virtual function.

```
class base {
private :
int x;
float y;
public:
virtual void display ( ) ;
int sum ( ) ;
};
class derivedD : public baseA
{
private :
int x ;
float y;
public:
void display ( ) ; //virtual
int sum ( ) ;
};
void main ( )
{
    baseA *ptr ;
    derivedD objd ;
    ptr = &objd ;
    Other Program statements
    ptr->display ( ) ; //run time binding
    ptr->sum ( ) ; //compile time binding
}
```

Note that the keyword virtual is followed by the return type of a member function if a run time is to be bound. Otherwise, the compile time binding will be effected as usual. In the above program segment, only the display () function has been declared as virtual in the base class, whereas the sum () is nonvirtual. Even though the message is given from the pointer of the base class to the objects of the derived class, it will not

access the sum () function of the derived class as it has been declared as nonvirtual. The sum () function compiles only the static binding.

The following program demonstrates the run time binding of the member functions of a class. The same message is given to access the derived class member functions from the array of pointers. As function are declared as virtual, the C++ compiler invokes the dynamic binding.

```

#include <iostream.h>
#include <conio.h>
class baseA {
public :
virtual void display () {
cout<< "One \n";
}
};
class derivedB : public baseA
{
    public:
    virtual void display(){
        cout<< "Two\n"; }
};
class derivedC: public derivedB
{
    public:
    virtual void display ( ) {
        cout<< "Three \n"; }
};
void main ( ) {
    //define three objects
    baseA obja;
    derivedB objb;
    derivedC objc;
    base A *ptr [3]; //define an array of pointers to baseA
    ptr [0] = &obja;
    ptr [1] = &objb;
    ptr [2] = &objc;
    for ( int i = 0; i <=2; i ++ )
        ptr [i]->display ( ); //same message for all objects
    getch ( );
}

```

Output

```

One
Two
Three

```

The program listed below illustrates the static binding of the member functions of a class. In program there are two classes student and academic. The class academic is derived from class student. The two member function getdata and display are defined for both the classes. *obj is defined for class student, the address of which is stored in the object of the class academic. The functions getdata () and display () of student class are invoked by the pointer to the class.

```

#include<iostream.h>
#include<conio.h>
class student {
private:
int rollno;
char name [20];
public:
void getdata ( );
void display ( );

```

```

};
class academic: public student {
private:
char stream;
public:
void getdata ( );
void display ( ) ;
};
void student:: getdata ( )
{
    cout<< "enter rollno\n";
    cin>>rollno;
    cout<< "enter name \n";
    cin>>name;
}
void student:: display ( )
{
    cout<< "the student's roll number is "<<rollno<< "and name is"<<name ;
    cout<< endl;
}
void academic :: getdata ( )
{
    cout<< "enter stream of a student? \n";
    cin >>stream;
}
void academic :: display ( ) {
    cout<< "students stream \n";
    cout <<stream<< endl;
}
void main ( )
{
    student *ptr ;
    academic obj ;
    ptr=&obj;
    ptr->getdata ( ) ;
    ptr->display ( ) ;
    getche ( ) ;
}
output
enter rollno
25
enter name
raghu
the student's roll number is 25 and name is raghu

```

The program listed below illustrates the dynamic binding of member functions of a class. In this program there are two classes student and academic. The class academic is derived from student. Student function has two virtual functions getdata () and display (). The pointer for student class is defined and object . for academic class is created. The pointer is assigned the address of the object and function of derived class are invoked by pointer to student.

```

#include <iostream.h>
#include <conio.h>
class student {

```

```

private:
introllno;
char name [20];
public:
virtual void getdata ( );
virtual void display ( );
};
class academic: public student {
private :
char stream[10];
public:
void getdata { };
void display ( ) ;
};
void student: : getdata ( )
{
    cout<< "enter rollno\n";
    cin >> rollno;
    cout<< "enter name \n";
    cin >> name;
}
void student:: display ( )
{
    cout<< "the student's roll number is"<<rollno<< "and name is"<<name;
    cout<< endl;
}
void academic: : getdata ( )
{
    cout << "enter stream of a student? \n";
    cin>> stream;
}
void academic:: display ( )
{
    cout<< "students stream \n";
    cout<< stream << endl;
}
void main ( )
{
    student *ptr ;
    academic obj ;
    ptr = &obj ;
    ptr->getdata ( );
    ptr->dlsplay ( );
    getch ( );
}

```

output
 enter stream of a student?
 Btech
 students stream
 Btech

LECTURE-32

Pure Virtual Functions

Generally a function is declared virtual inside a base class and we redefine it the derived classes. The function declared in the base class seldom performs any task.

The following program demonstrates how a pure virtual function is defined, declared and invoked from the object of a derived class through the pointer of the base class. In the example there are two classes employee and grade. The class employee is base class and the grade is derived class. The functions getdata () and display () are declared for both the classes. For the class employee the functions are defined with empty body or no code inside the function. The code is written for the grade class. The methods of the derived class are invoked by the pointer to the base class.

```
#include<iostream.h>
#include<conio.h>
class employee {
int code
char name [20] ;
public:
virtual void getdata ( ) ;
virtual void display ( ) ;
};
class grade: public employee
{
    char grd [90] ;
    float salary ;
public :
    void getdata ( ) ;
    void display ( ) ;
};
void employee :: getdata ( )
{
}
void employee:: display ( )
{
}
void grade : : getdata ( )
{
    cout<< " enter employee's grade ";
    cin>> grd ;
    cout<< "\n enter the salary " ;
    cin>> salary;
}
void grade : : display ( )
{
    cout<<" Grade salary \n";
    cout<< grd<< " "<< salary<< endl;
```

```

    }
    void main ( )
    {
        employee *ptr ;
        grade obj ;
        ptr = &obj ;
        ptr->getdata ( ) ;
        ptr->display ( ) ;
        getche ( ) ;
    }

```

Output

```

enter employee's grade A
enter the salary 250000
Grade salary
A      250000

```

Object Slicing:

In C++, a derived class object can be assigned to a base class object, but the other way is not possible.

```

class Base { int x, y; };

class Derived : public Base { int z, w; };

int main()
{
    Derived d;
    Base b = d; // Object Slicing, z and w of d are sliced off
}

```

Object Slicing happens when a derived class object is assigned to a base class object, additional attributes of a derived class object are sliced off to form the base class object.

```

#include <iostream>
using namespace std;

class Base
{
protected:
    int i;
public:
    Base(int a)      { i = a; }
    virtual void display()
    { cout << "I am Base class object, i = " << i << endl; }
};

class Derived : public Base
{
    int j;
public:
    Derived(int a, int b) : Base(a) { j = b; }
    virtual void display()
    { cout << "I am Derived class object, i = "
      << i << ", j = " << j << endl; }
};

```



```
// Global method, Base class object is passed by value
void somefunc (Base obj)
{
    obj.display();
}

int main()
{
    Base b(33);
    Derived d(45, 54);
    somefunc(b);
    somefunc(d); // Object Slicing, the member j of d is sliced off
    return 0;
}
```

Output:

```
I am Base class object, i = 33
I am Base class object, i = 45
```

We can avoid above unexpected behavior with the use of pointers or references. Object slicing doesn't occur when pointers or references to objects are passed as function arguments since a pointer or reference of any type takes same amount of memory. For example, if we change the global method myfunc() in the above program to following, object slicing doesn't happen.

```
// rest of code is similar to above
void somefunc (Base &obj)
{
    obj.display();
}
// rest of code is similar to above
```

Output:

```
I am Base class object, i = 33
I am Derived class object, i = 45, j = 54
```

We get the same output if we use pointers and change the program to following.

```
// rest of code is similar to above
void somefunc (Base *objp)
{
    objp->display();
}
int main()
{
    Base *bp = new Base(33) ;
    Derived *dp = new Derived(45, 54);
    somefunc(bp);
    somefunc(dp); // No Object Slicing
    return 0;
}
```

Output:

```
I am Base class object, i = 33
I am Derived class object, i = 45, j = 54
```

Object slicing can be prevented by making the base class function pure virtual there by disallowing object creation. It is not possible to create the object of a class which contains a pure virtual method.

LECTURE-33

C++ Function Overriding

If base class and derived class have member functions with same name and arguments. If you create an object of derived class and write code to access that member function then, the member function in derived class is only invoked, i.e., the member function of derived class overrides the member function of base class. This feature in C++ programming is known as function overriding.

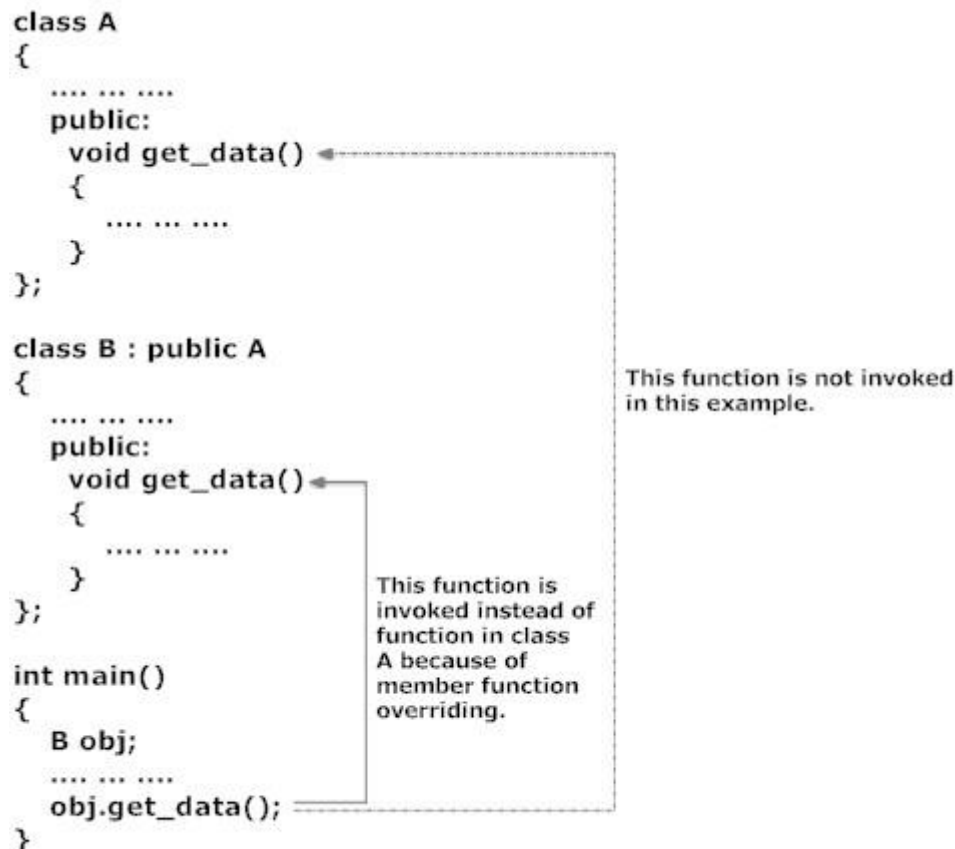


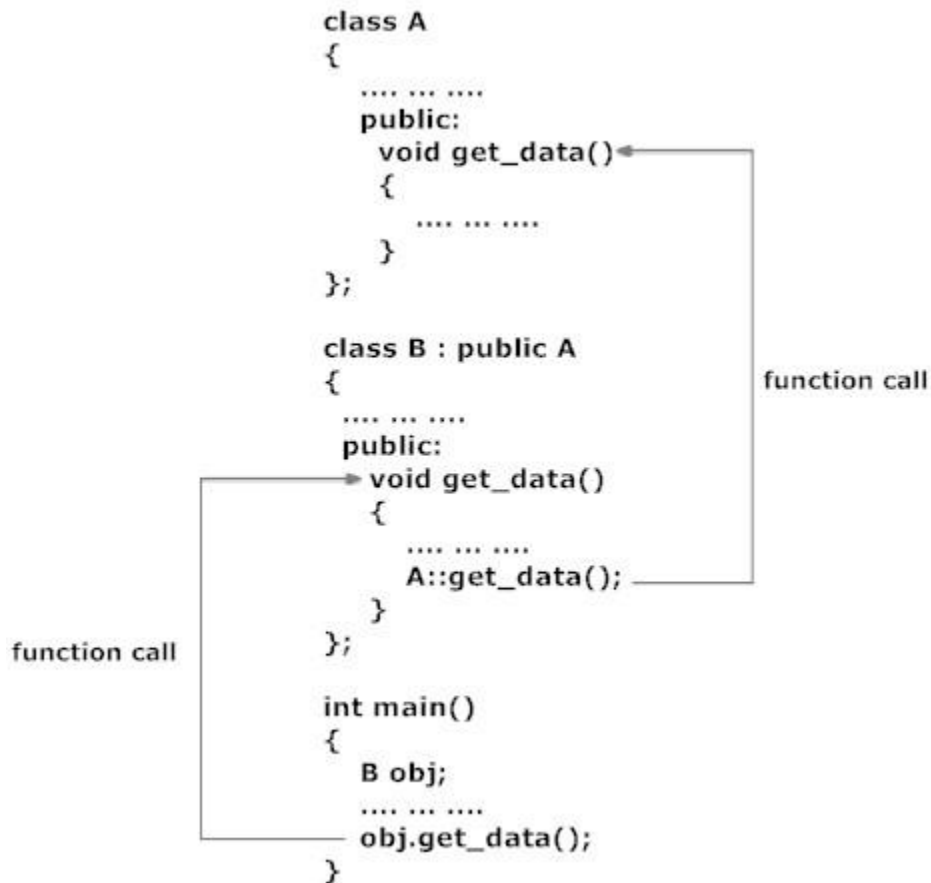
Figure: Member Function Overriding in C++

Accessing the Overridden Function in Base Class From Derived Class

To access the overridden function of base class from derived class, scope resolution operator `::`. For example: If you want to access `get_data()` function of base class from derived class in above example then, the following statement is used in derived class.

```
A::get_data; // Calling get_data() of class A.
```

It is because, if the name of class is not specified, the compiler thinks `get_data()` function is calling itself.



Abstract Class

Abstract Class is a class which contains atleast one Pure Virtual function in it. Abstract classes are used to provide an Interface for its sub classes. Classes inheriting an Abstract Class must provide definition to the pure virtual function, otherwise they will also become abstract class.

Characteristics of Abstract Class

1. Abstract class cannot be instantiated, but pointers and references of Abstract class type can be created.
2. Abstract class can have normal functions and variables along with a pure virtual function.
3. Abstract classes are mainly used for Upcasting, so that its derived classes can use its interface.
4. Classes inheriting an Abstract Class must implement all pure virtual functions, or else they will become Abstract too.

Pure Virtual Functions

Pure virtual Functions are virtual functions with no definition. They start with **virtual** keyword and ends with = 0. Here is the syntax for a pure virtual function,

```
virtual void f() = 0;
```

Example of Abstract Class

```
class Base    //Abstract base class
```

```

{
public:
virtual void show() = 0;      //Pure Virtual Function
};

class Derived:public Base
{
public:
void show()
{ cout << "Implementation of Virtual Function in Derived class"; }
};

int main()
{
Base obj;    //Compile Time Error
Base *b;
Derived d;
b = &d;
b->show();
}

```

Output : Implementation of Virtual Function in Derived class

In the above example Base class is abstract, with pure virtual **show()** function, hence we cannot create object of base class.

Why can't we create Object of Abstract Class ?

When we create a pure virtual function in Abstract class, we reserve a slot for a function in the VTABLE(studied in last topic), but doesn't put any address in that slot. Hence the VTABLE will be incomplete.

As the VTABLE for Abstract class is incomplete, hence the compiler will not let the creation of object for such class and will display an error message whenever you try to do so.

LECTURE-34

Exception Handling:

Exception refers to unexpected condition in a program. The unusual conditions could be faults, causing an error which in turn causes the program to fail. The error handling mechanism of c++ is generally referred to as exception handling.

Generally , exceptions are classified into synchronous and asynchronous exceptions.. The exceptions which occur during the program execution, due to some fault in the input data or technique that is not suitable to handle the current class of data. with in a program is known as synchronous exception.

Example:

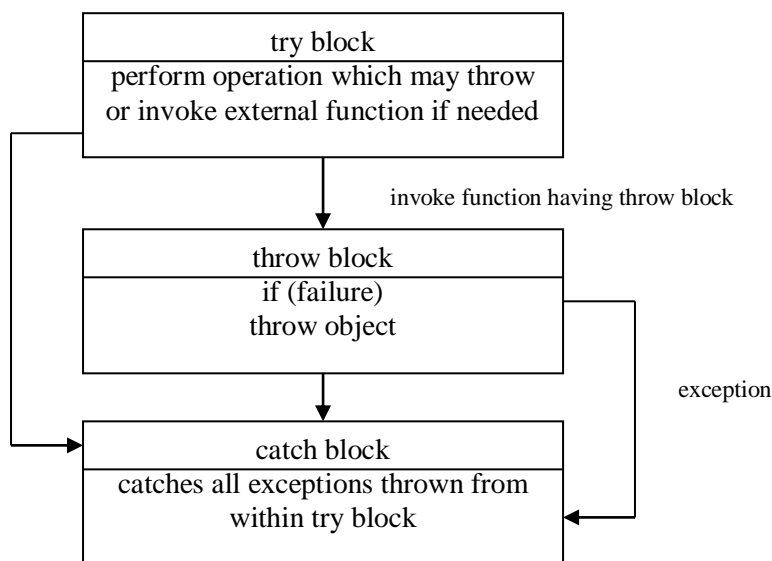
errors such as out of range,overflow,underflow and so on.

The exceptions caused by events or faults unrelated to the program and beyond the control of program are asynchronous exceptions.

For example, errors such as keyboard interrupts, hardware malfunctions, disk failure and so on.

exception handling model:

When a program encounters an abnormal situation for which it is not designed, the user may transfer control to some other part of the program that is designed to deal with the problem. This is done by throwing an exception. The exception handling mechanism uses three blocks: try, throw and catch. The try block must be followed immediately by a handler, which is a catch block. If an exception is thrown in the try block the program control is transferred to the appropriate exception handler. The program should attempt to catch any exception that is thrown by any function. The relationship of these three exceptions handling constructs called the exception handling model is shown in figure:



throw construct:

The keyword throw is used to raise an exception when an error is generated in the computation. the throw expression initialize a temporary object of the type T used in throw (T arg).

syntax:

```
throw T;
```

catch construct:

The exception handler is indicated by the catch keyword. It must be used immediately after the statements marked by the try keyword. The catch handler can also occur immediately after another catch. Each handler will only evaluate an exception that matches.

syn:

```
catch(T)
{
// error messages
}
```

try construct:

The try keyword defines a boundary within which an exception can occur. A block of code in which an exception can occur must be prefixed by the keyword try. Following the try keyword is a block of code enclosed by braces. This indicates that the prepared to test for the existence of exceptions. If an exception occurs, the program flow is interrupted.

```
try
{
...
if (failure)
    throw T;
}
catch(T)
{
...
}
```

example:

```
#include<iostream.h>
void main()
{
int a,b;
cout<<"enter two numbers:";
cin>>a>>b;
try
{
if (b==0)
    throw b;
else
    cout<<a/b;
}
catch(int x)
{
cout<<"2nd operand can't be 0";
}
}
```

LECTURE-35

Array reference out of bound:

```
#define max 5
class array
{
private:
    int a[max];
public:
    int &operator[](int i)
    {
        if (i<0 || i>=max)
            throw i;

        else
            return a[i];
    }
};

void main()
{
    array x;
    try
    {
        cout<<"trying to refer a[1]..."
        x[1]=3;
        cout<<"trying to refer a[13]..."
        x[13]=5;
    }
    catch(int i)
    {
        cout<<"out of range in array references...";
    }
}
```

multiple catches in a program

```
void test(int x)
{
    try{
        if (x==1)
            throw x;
        else if (x==-1)
            throw 3.4;
        else if (x==0)
            throw 's';
    }
    catch (int i)
    {
        cout<<"caught an integer...";
    }
    catch (float s)
    {
        cout<<"caught a float...";
    }
}
```

```

}
catch (char c)
{
cout<<"caught a character...";
}}
void main()
{
test(1);
test(-1);
test(0);
}

```

catch all

```

void test(int x)
{
try{
if (x==1)
    throw x;
else if (x==-1)
    throw 3.4;
else if (x==0)
    throw 's';
}
catch (...)
{
cout<<"caught an error...";
}
}

```


Module-03:
LECTURE-36

Containership in C++

When a class contains objects of another class or its members, this kind of relationship is called containership or nesting and the class which contains objects of another class as its members is called as container class.

Syntax for the declaration of another class is:

Class class_name1

{

};

Class class_name2

{

};

Class class_name3

{

Class_name1 obj1; // object of class_name1

Class_name2 obj2; // object of class_name2

};

```

//Sample Program to demonstrate Containership
#include < iostream.h >
#include < conio.h >
#include < iomanip.h >
#include< stdio.h >
const int len=80;
class employee
{
private:
char name[len];
int number;
public:
void get_data()
{
cout << "\n Enter employee name: ";
cin >> name;
cout << "\n Enter employee number: ";
cin >> number;
}
void put_data()
{
cout << " \n\n Employee name: " << name;
cout << " \n\n Employee number: " << number;
}
};
class manager
{
private:
char dept[len];
int numemp;
employee emp;
public:
void get_data()
{
emp.get_data();
cout << " \n Enter department: ";
cin >> dept;
cout << "\n Enter number of employees: ";
cin >> numemp;
}
void put_data()
{
emp.put_data();
cout << " \n\n Department: " << dept;
cout << " \n\n Number of employees: " << numemp;
}
};
class scientist
{
private:
int pubs,year;
employee emp;
public:

```

```

void get_data()
{
    emp.get_data();
    cout << " \n Number of publications: ";
    cin >> pubs;
    cout << " \n Year of publication: ";
    cin >> year;
}
void put_data()
{
    emp.put_data();
    cout << "\n\n Number of publications: " << pubs;
    cout << "\n\n Year of publication: " << year;
}
};
void main()
{
    manager m1;
    scientist s1;
    int ch;
    clrscr();
    do
    {
        cout << "\n 1.manager\n 2.scientist\n";
        cout << "\n  Enter your choice: ";
        cin >> ch;
        switch(ch)
        {
            case 1:
                cout << "\n Manager data:\n";
                m1.get_data();
                cout << "\n Manager data:\n";
                m1.put_data();
                break;
            case 2:cout << " \n Scientist data:\n";
                s1.get_data();
                cout << " \n Scientist data:\n";
                s1.put_data();
                break;
        }
        cout << "\n\n To continue Press 1 -> ";
        cin >> ch;
    }
    while(ch==1);
    getch();
}

```

Difference between Inheritance and Containership :

Containership: Containership is the phenomenon of using one or more classes within the definition of other class. When a class contains the definition of some other classes, it is referred to as composition, containment or aggregation. The data member of a new class is an object of some other class. Thus the other class is said to be composed of other classes and hence referred to as containership. Composition is often referred to as a “has-a” relationship because the objects of the composite class have objects of the composed class as members.

Inheritance: Inheritance is the phenomenon of deriving a new class from an old one. Inheritance supports code reusability. Additional features can be added to a class by deriving a class from it and then by adding new features to it. Class once written or tested need not be rewritten or redefined. Inheritance is also referred to as specialization or derivation, as one class is inherited or derived from the other. It is also termed as “is-a” relationship because every object of the class being defined is also an object of the inherited class.

LECTURE-37

Template:

Template supports generic programming, which allows developing reusable software components such as functions, classes, etc supporting different data types in a single frame work.

A template in c++ allows the construction of a family of template functions and classes to perform the same operation o different data types. The templates declared for functions are called class templates. They perform appropriate operations depending on the data type of the parameters passed to them.

Function Templates:

A function template specifies how an individual function can be constructed.

```
template <class T>
return type functionnm(T arg1,T arg2)
{
fn body;
}
```

For example:

Input two number and swap their values

```
template <class T>
void swap (T &x,T & y)
{
T z;
z=x;
x=y;
y=z;
}
void main( )
{
char ch1,ch2;
cout<<"enter two characters:";
cin>>ch1>>ch2;
swap(ch1,ch2);
cout<<ch1<<ch2;
int a,b;
cout<<"enter a,b:";
cin>>a>>b;
swap(a,b);
cout<<a<<b;
float p,q;
cout<<"enter p,q:";
cin>>p>>q;
swap(p,q);
cout<<p<<q;
}
```

example 2:

find maxium between two data items.

```
template <class T>
T max(T a,T b)
```

```

{
if (a>b)
return a;
else
return b;
}
void main()
{
char ch1,ch2;
cout<<"enter two characters:";
cin>>ch1>>ch2;
cout<<max(ch1,ch2);
int a,b;
cout<<"enter a,b:";
cin>>a>>b;
cout<<max(a,b);
float p,q;
cout<<"enter p,q:";
cin>>p>>q;
cout<<max(p,q);
}

```

Overloading of function template

```

#include<iostream.h>
template <class T>
void print( T a)
{
    cout<<a;
}
template <class T>
void print( T a, int n)
{
    int i;
    for (i=0;i<n;i++)
        cout<<a;
}
void main()
{
    print(1);
    print(3.4);
    print(455,3);
    print("hello",3);
}

```

Multiple arguments function template:

find sum of two different numbers

```

template <class T,class U>
T sum(T a,U b)
{
    return a+(U)b;
}
void main( )

```

```
{  
cout<<sum(4,5.5);  
cout<sum(5.4,3);  
}
```

LECTURE-38

Class Template

similar to functions, classes can also be declared to operate on different data types. Such classes are class templates. a class template specifies how individual classes can be constructed similar to normal class definition. These classes model a generic class which support similar operations for different data types.

syn:

```
template <class T>
class classnm
{
T member1;
T member2;
...
...
public:
T fun();
...
..
};
```

objects for class template is created like:

```
classnm <datatype> obj;
obj.memberfun();
```

example:

Input n numbers into an array and print the element is ascending order.(array sorting)

```
template <class T>
class array
{
T *a;
int n;
public:
void getdata()
{
int i;
cout<<"enter how many no:";
cin>>n;
a=new T[n];
for (i=0;i<n;i++)
{
cout<<"enter a number:";
cin>>a[i];
}
}
void putdata()
{
```



```

for (i=0;i<n;i++)
{
cout<<a[i]<<endl;
}
}
void sort( )
{
T k;
int i,j;
for(i=0;i<n-1;i++)
{
    for (j=0;j<n;j++)
    {
        if (a[i]>a[j])
        {
            k=a[i];
            a[i]=a[j];
            a[j]=k;
        }
    }
}
};
void main()
{
array <int>x;
x.getdata();
x.sort();
x.putdata();

array <float> y;
y.getdata();
y.sort();
y.putdata();
}

```

LECTURE-39

Virtual destructors:

Just like declaring member functions as virtual, destructors can be declared as virtual, whereas constructors can not be virtual. Virtual Destructors are controlled in the same way as virtual functions. When a derived object pointed to by the base class pointer is deleted, destructor of the derived class as well as destructor of all its base classes are invoked. If destructor is made as non virtual destructor in the base class, only the base class's destructor is invoked when the object is deleted.

```
#include<iostream.h>
#include<string.h>
class father
{
protected:
char *fname;
public:
father(char *name)
{
fname=new char(strlen(name)+1);
strcpy(fname,name);
}
virtual ~father()
{
delete fname;
cout<<"~father is invoked...";
}

virtual void show()
{
cout<<"father name..."<<fname;
}
};

class son: public father
{
protected:
char *s_name;
public:
son(char *fname,char *sname):father(fname)
{
sname=new char[strlen(sname)+1];
strcpy(s_name,sname);
}
~son()
{
delete s_name;
cout<<"~son() is invoked"<<endl;
}
void show()
{
cout<<"father's name"<<fname;
cout<<"son's name:"<<s_name;
```

```

}
};
void main()
{
father *basep;
basep =new father ("mona");
cout<<"basep points to base object..."
basep->show();
delete basep;
basep=new son("sona","mona");
cout<<"base points to derived object...";
basep->show();
delete basep;
}

```

Overloading of >> and << operator

```

#define size 5
class vector
{
int v[size];
public:
vector();
friend vector operator*(int a,vector b);
friend vector operator *(vector b,int a);
friend istream &operator>>(istream &,vector &);
friend ostream &operator<<(ostream &,vector &);
};
vector :: vector()
{
for(int i=0;i<size;i++)
v[i]=0;
}
vector::vector(int *x)
{
for (int i=0;i<size;i++)
v[i]=x[i];
}
vector operator*(int a,vector b)
{
vector c;
for(int i=0;i<size;i++)
c.v[i]=a*b.v[i];
return c;
}

vector operator*(vector b,int a)
{
vector c;

```

```

for(int i=0;i<size;i++)
c.v[i]=a*b.v[i];
return c;
}
istream &operator>>(istream &din,vector &b)
{
for(int i=0;i<size;i++)
din>>b.v[i];
}
ostream &operator<<(ostream &dout,vector &b)
{
for(i=0;i<size;i++)
dout<<a[i];
return dout;
}
int x[size]={2,4,6};
int main()
{
vector m;
vector n=x;
cout<<"enter elements of vector m";
cin>>m;
cout<<m;
vector p,q;
p=2*m;
q=n*2;
cout<<p;
cout<<q;

}

```

LECTURE-40

Managing Console I/O

Introduction

One of the most essential features of interactive programming is its ability to interact with the users through operator console usually comprising keyboard and monitor. Accordingly, every computer language (and compiler) provides standard input/output functions and/or methods to facilitate console operations.

C++ accomplishes input/output operations using concept of stream. A stream is a series of bytes whose value depends on the variable in which it is stored. This way, C++ is able to treat all the input and output operations in a uniform manner. Thus, whether it is reading from a file or from the keyboard, for a C++ program it is simply a stream.

We have used the objects `cin` and `cout` (pre-defined in the `iostream.h` file) for the input and output of data of various types. This has been made possible by overloading the operators `>>` and `<<` to recognize all the basic C++ types. The `>>` operator is overloaded in the `istream` class and `<<` is overloaded in the `ostream` class. The following is the general format for reading data from the keyboard:

```
cin >> variable1 >> variable2 >>... ..>> variableN;
```

Where `variable1`, `variable2`,.... are valid C++ variable names that have been declared already. This statement will cause the computer to halt the execution and look for input data from the keyboard. The input data for this statement would be:

```
data1 data2.....dataN
```

The input data are separated by white spaces and should match the type of variable in the `cin` list. Spaces, newlines and tabs will be skipped.

The operator `>>` reads the data character by character and assigns it to the indicated location. The reading for a variable will be terminated at the encounter of a white space or a character that does not match the destination type.

For example, consider the following code:

```
int code;  
cin >> code;
```

Suppose the following data is given as input:

```
1267E
```

The operator will read the characters up to 7 and the value 1267 is assigned to `code`. The character `E` remains in the input stream and will be input to the next `cin` statement. The general format of outputting data:

```
cout << item1 << item2 << .. ..<< itemN;
```

The items, `item1` through `itemN` may be variables or constants of any basic types.

The put() and get() Functions

The classes istream and ostream define two member functions get() and put() respectively to handle the single character input/output operations. There are two types of get() functions. We can use both get(char*) and get(void) prototypes to fetch a character including the blank space, tab and the newline character. The get(char*) version assigns the input character to its argument and the get(void) version returns the input character.

Since these functions are members of the input/output stream classes, we must invoke them using an appropriate object. For instance, look at the code snippet given below:

```
char c;
cin.get (c); //get a character from keyboard and assign it to c
while (c!= '\n')
{
    cout << C;    //display the character on screen cin.get (c);
                  //get another character
}
```

This code reads and displays a line of text (terminated by a newline character). Remember, the operator >> can also be used to read a character but it will skip the white spaces and newline character. The above while loop will not work properly if the statement

```
cin >> c;
is used in place of
cin.get (c);
```

Try using both of them and compare the results. The get(void) version is used as follows:

```
char c;
c - cin.getl(); //cin.get (c) replaced
```

The value returned by the function get() is assigned to the variable c.

The function put(), a member of ostream class, can be used to output a line of text, character by character. For example,

```
cout << put ('x');
displays the character x and
cout << put (ch);
displays the value of variable ch.
```

The variable ch must contain a character value. We can also use a number as an argument to the function put (). For example,

```
cout << put (68);
```

displays the character D. This statement will convert the int value 90 to a char value and display the character whose ASCII value is 68,

The following segment of a program reads a line of text from the keyboard and displays it on the screen.

```
char c;
cin.get (c)    //read a character
while (c!= '\n')
{
    cout<< put(c); //display the character on screen cin.get (c) ;
}
```

The getline () and write () Functions

We can read and display a line of text more efficiently using the line-oriented input/output functions getline() and write(). The getline() function reads a whole line of text that ends with a newline character. This function can be invoked by using the object cin as follows:

```
cin.getline(line, size);
```

This function call invokes the function which reads character input into the variable line. The reading is terminated as soon as either the newline character '\n' is encountered or size number of characters are read (whichever occurs first). The newline character is read but not saved. Instead, it is replaced by the null character.

For example; consider the following code:

```
char name [20] ;
```

```
cin.getline(name, 20);
```

Assume that we have given the following input through the keyboard:

Neeraj good

This input will be read correctly and assigned to the character array name. Let us suppose the input is as follows:

Object Oriented Programming

In this case, the input will be terminated after reading the following 19 characters:

Object Oriented Pro

After reading the string/ cin automatically adds the terminating null character to the character array.

Remember, the two blank spaces contained in the string are also taken into account, i.e. between Objects and Oriented and Pro.

We can also read strings using the operator >> as follows:

```
cin >> name;
```

But remember cin can read strings that do not contain white space. This means that cin can read just one word and not a series of words such as “Neeraj good”.

Formatted Console I/O Operations

C++ supports a number of features that could be used for formatting the output. These features include:

- ios class functions and flags.
- Manipulators.
- User-defined output functions.

The ios class contains a large number of member functions that could be used to format the output in a number of ways. The most important ones among them are listed below.

Table 10.1

Function	Task
width()	To specify the required field size for displaying an output value
Precision()	To specify the number of digits to be displayed after the decimal point of a float value
fill()	To specify a character that is used to fill the unused portion of a field.
self()	To specify format flags that can control the form of output display (such as Left-justification and right-justification).
Unself()	To clear the flags specified.

Manipulators are special functions that can be included in these statements to alter the format parameters of a stream. The table given below shows some important! manipulator functions that are frequently used. To access these manipulators, the file `iomanip.h` should be included in the program.

Table 10.2

Manipulator	Equivalent Ios function
<code>setw()</code>	<code>width()</code>
<code>Setprecision()</code>	<code>Precision()</code>
<code>Setfill()</code>	<code>fill()</code>
<code>setiosflags()</code>	<code>self()</code>
<code>Resetiosflags()</code>	<code>Unself()</code>

In addition to these functions supported by the C++ library, we can create our own manipulator functions to provide any special output formats.

Student Activity

1. What is a stream?
2. Define `put ()` and `get ()` functions
3. What is the difference between `getline ()` and `get ()` functions?
4. Define `write ()` functions.
5. What are manipulators?

Streams

C++ is designed to work with a wide variety of devices including terminals, disks, and tape drives. Although each device is very different, the system supplies an interface to the programmer that is independent of the actual device being accessed, This interface is known as stream.

A stream is a sequence of bytes. It acts either as a source from which the input data can be obtained or as a destination to which the output data can be sent. The source stream that provides data to the program is called the input stream. In other words, a program extracts the bytes from an input stream and inserts bytes into an output stream.

The data in the input stream can come from the keyboard or any other storage device. Similarly, the data in the output stream can go to the screen or any other storage device. As mentioned earlier, a stream acts as an interface between the program and the input/output device. Therefore, a C++ program handles data (input or output) independent of the devices used.

C++ contains several pre-defined streams that are automatically opened when a program begins its execution. They include `cin` and `cout` which have been used very often in our earlier programs. We know that `cin` represents the input stream connected to the standard input device (usually the keyboard) and `cout` represents the output stream connected to the standard output device (usually the screen). Note that the keyboard and the screen are default options. We can redirect streams to other devices or files, if necessary.

I/O Operations

Input and Output statements of computer languages are used to provide communications between the user and the program. In most of the computer languages, input and output are done

through statements. But in C++, these operations are carried out through its built-in functions. The I/O functions are designed in header files like fstream.h, iostream.h etc.

Through these functions, data can be read from or written to files or standard input/output devices like keyboard and VDU. This execution of a program can be interrupted by input/output calls. Hence the data can be entered or output can be retrieved during execution.

The file, stream classes support a number of member functions for performing the input and output operations on files. One pair of functions, put() and get(), are designed for handling a single character at a time. Another pair of functions,

write() and read(), are designed to write and read blocks of binary data.

put() and get() Functions

The function put() writes a single character to the associated stream. Similarly, the function get() reads a single character from the, associated stream. The program, requests for a string. On receiving the string, the program writes it, character, by character, to the file using the put() function in a for loop. Note that the length of the string is used to terminate the for loop.

C++ provides a number of useful predefined stream classes for console input/output operations. Some of the C++ the predefined stream objects are listed below.

- cin This is the name of standard input stream, usually keyboard. The corresponding name in C is stdin.
- cout This is the name of standard output stream, usually screen of the monitor. The corresponding name in C is stdout.
- cerr This is the name of standard error output stream, usually screen of the monitor. The corresponding name in C is stderr.
- clog This is another version of cerr. It provides buffer to collect errors. C does not have a stream equivalent to this.

In their default roles, these streams are tied up with the keyboard and screen of the monitor as describe above. However, you can redirect them from and to other devices and files.

Keywords

put (): A member of ostream class, could be used to output a line of text, character by character.

Get (): A member of istream class, used to input a single character at a line.

Getline (): The get line () function reads a whole line of -text that ends with a new line character. This function could be invoked by using the object cin.

Manipulators: Special functions that can be included in console I/O statements to alter the format-parameters of a stream

Streams: C++ is designed to work with a wide variety of devices including, disks and tape drives. Although each device is very different the system supplies an interface to the programmer that is independent of the actual device accessed. This interface is known as stream.

Output stream: The source stream that provides data to the program is called the output stream.

LECTURE-41

Namespaces :

Scopes

Named entities, such as variables, functions, and compound types need to be declared before being used in C++. The point in the program where this declaration happens influences its visibility:

An entity declared outside any block has *global scope*, meaning that its name is valid anywhere in the code. While an entity declared within a block, such as a function or a selective statement, has *block scope*, and is only visible within the specific block in which it is declared, but not outside it.

Variables with block scope are known as *local variables*.

For example, a variable declared in the body of a function is a *local variable* that extends until the end of the the function (i.e., until the brace } that closes the function definition), but not outside it:

```
int foo;           // global variable

int some_function ()
{
    int bar;       // local variable
    bar = 0;
}

int other_function ()
{
    foo = 1;  // ok: foo is a global variable
    bar = 2;  // wrong: bar is not visible from this function
}
```

In each scope, a name can only represent one entity. For example, there cannot be two variables with the same name in the same scope:

```
int some_function ()
{
    int x;
    x = 0;
    double x;  // wrong: name already used in this scope
    x = 0.0;
}
```

The visibility of an entity with *block scope* extends until the end of the block, including inner blocks. Nevertheless, an inner block, because it is a different block, can re-utilize a name existing in an outer scope to refer to a different entity; in this case, the name will refer to a different entity only within the inner block, hiding the entity it names outside. While outside it, it will still refer to the original entity. For example:

```
// inner block scopes
#include <iostream>
```

```
using namespace std;

int main () {
    int x = 10;
    int y = 20;
    {
        int x;    // ok, inner scope.
        x = 50;   // sets value to inner x
        y = 50;   // sets value to (outer) y
        cout << "inner block:\n";
        cout << "x: " << x << '\n';
        cout << "y: " << y << '\n';
    }
    cout << "outer block:\n";
    cout << "x: " << x << '\n';
    cout << "y: " << y << '\n';
    return 0;
}
```

output:

```
inner block:
x: 50
y: 50
outer block:
x: 10
y: 50
```

Note that `y` is not hidden in the inner block, and thus accessing `y` still accesses the outer variable.

Variables declared in declarations that introduce a block, such as function parameters and variables declared in loops and conditions (such as those declared on a `for` or an `if`) are local to the block they introduce.

Namespaces

Only one entity can exist with a particular name in a particular scope. This is seldom a problem for local names, since blocks tend to be relatively short, and names have particular purposes within them, such as naming a counter variable, an argument, etc...

But non-local names bring more possibilities for name collision, especially considering that libraries may declare many functions, types, and variables, neither of them local in nature, and some of them very generic.

Namespaces allow us to group named entities that otherwise would have *global scope* into narrower scopes, giving them *namespace scope*. This allows organizing the elements of programs into different logical scopes referred to by names.

The syntax to declare a namespaces is:

```
namespace identifier
{
    named_entities
}
```

Where `identifier` is any valid identifier and `named_entities` is the set of variables, types and functions that are included within the namespace. For example:

```
namespace myNamespace
{
    int a, b;
}
```

In this case, the variables `a` and `b` are normal variables declared within a namespace called `myNamespace`.

These variables can be accessed from within their namespace normally, with their identifier (either `a` or `b`), but if accessed from outside the `myNamespace` namespace they have to be properly qualified with the scope operator `::`. For example, to access the previous variables from outside `myNamespace` they should be qualified like:

```
1 myNamespace::a
2 myNamespace::b
```

Namespaces are particularly useful to avoid name collisions. For example:

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

namespace foo
{
    int value() { return 5; }
}

namespace bar
{
    const double pi = 3.1416;
    double value() { return 2*pi; }
}

int main () {
    cout << foo::value() << '\n';
    cout << bar::value() << '\n';
    cout << bar::pi << '\n';
    return 0;
}
```

output:

```
5
6.2832
3.1416
```

In this case, there are two functions with the same name: `value`. One is defined within the namespace `foo`, and the other one in `bar`. No redefinition errors happen thanks to namespaces. Notice also how `pi` is accessed in an unqualified manner from within namespace `bar` (just as `pi`), while it is again accessed in `main`, but here it needs to be qualified as `bar::pi`.

Namespaces can be split: Two segments of a code can be declared in the same namespace:

```
1 namespace foo { int a; }
```

```
2 namespace bar { int b; }
3 namespace foo { int c; }
```

This declares three variables: `a` and `c` are in namespace `foo`, while `b` is in namespace `bar`. Namespaces can even extend across different translation units (i.e., across different files of source code).

using

The keyword `using` introduces a name into the current declarative region (such as a block), thus avoiding the need to qualify the name. For example:

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

namespace first
{
    int x = 5;
    int y = 10;
}

namespace second
{
    double x = 3.1416;
    double y = 2.7183;
}

int main () {
    using first::x;
    using second::y;
    cout << x << '\n';
    cout << y << '\n';
    cout << first::y << '\n';
    cout << second::x << '\n';
    return 0;
}
```

Output:

```
5
2.7183
10
3.1416
```

Notice how in `main`, the variable `x` (without any name qualifier) refers to `first::x`, whereas `y` refers to `second::y`, just as specified by the `using` declarations. The variables `first::y` and `second::x` can still be accessed, but require fully qualified names.

The keyword `using` can also be used as a directive to introduce an entire namespace:

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

namespace first
{
    int x = 5;
    int y = 10;
```

```

}

namespace second
{
    double x = 3.1416;
    double y = 2.7183;
}

int main () {
    using namespace first;
    cout << x << '\n';
    cout << y << '\n';
    cout << second::x << '\n';
    cout << second::y << '\n';
    return 0;
}

```

output:

```

5
10
3.1416
2.7183

```

In this case, by declaring that we were using namespace `first`, all direct uses of `x` and `y` without name qualifiers were also looked up in namespace `first`.

`using` and `using namespace` have validity only in the same block in which they are stated or in the entire source code file if they are used directly in the global scope. For example, it would be possible to first use the objects of one namespace and then those of another one by splitting the code in different blocks:

```

// using namespace example
#include <iostream>
using namespace std;

namespace first
{
    int x = 5;
}

namespace second
{
    double x = 3.1416;
}

int main () {
    {
        using namespace first;
        cout << x << '\n';
    }
    {
        using namespace second;
        cout << x << '\n';
    }
    return 0;
}

```

output:

```

5
3.1416

```

Namespace aliasing

Existing namespaces can be aliased with new names, with the following syntax:

```
namespace new_name = current_name;
```

The std namespace

All the entities (variables, types, constants, and functions) of the standard C++ library are declared within the `std` namespace. Most examples in these tutorials, in fact, include the following line:

```
using namespace std;
```

This introduces direct visibility of all the names of the `std` namespace into the code. This is done in these tutorials to facilitate comprehension and shorten the length of the examples, but many programmers prefer to qualify each of the elements of the standard library used in their programs. For example, instead of:

```
cout << "Hello world!";
```

It is common to instead see:

```
std::cout << "Hello world!";
```

Whether the elements in the `std` namespace are introduced with `using` declarations or are fully qualified on every use does not change the behavior or efficiency of the resulting program in any way. It is mostly a matter of style preference, although for projects mixing libraries, explicit qualification tends to be preferred.

Storage classes

The storage for variables with *global* or *namespace scope* is allocated for the entire duration of the program. This is known as *static storage*, and it contrasts with the storage for *local variables* (those declared within a block). These use what is known as automatic storage. The storage for local variables is only available during the block in which they are declared; after that, that same storage may be used for a local variable of some other function, or used otherwise.

But there is another substantial difference between variables with *static storage* and variables with *automatic storage*:

- Variables with *static storage* (such as global variables) that are not explicitly initialized are automatically initialized to zeroes.
- Variables with *automatic storage* (such as local variables) that are not explicitly initialized are left uninitialized, and thus have an undetermined value.

For example:

```
// static vs automatic storage  
#include <iostream>
```

```
using namespace std;

int x;

int main ()
{
    int y;
    cout << x << '\n';
    cout << y << '\n';
    return 0;
}
```

Output:

```
0
4285838
```

The actual output may vary, but only the value of `x` is guaranteed to be zero. `y` can actually contain just about any value (including zero).

Lecture-42:

New & Delete Operators

Dynamic memory allocation means creating memory at runtime. For example, when we declare an array, we must provide size of array in our source code to allocate memory at compile time.

But if we need to allocate memory at runtime we must use new operator followed by data type. If we need to allocate memory for more than one element, we must provide total number of elements required in square bracket []. It will return the address of first byte of memory.

Syntax of new operator

```
ptr = new data-type;  
//allocate memory for one element  
  
ptr = new data-type [ size ];  
//allocate memory for fixed number of element
```

Delete operator is used to deallocate the memory created by new operator at run-time. Once the memory is no longer needed it should be freed so that the memory becomes available again for other request of dynamic memory.

Syntax of delete operator

```
delete ptr;  
//deallocate memory for one element  
  
delete[] ptr;  
//deallocate memory for array
```

Example of c++ new and delete operator

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

```

void main()
{

    int size,i;
    int *ptr;

    cout<<"\n\tEnter size of Array : ";
    cin>>size;

    ptr = new int[size];
    //Creating memory at run-time and return first byte of address to ptr.
    for(i=0;i<5;i++)    //Input array from user.
    {
        cout<<"\nEnter any number : ";
        cin>>ptr[i];
    }
    for(i=0;i<5;i++)    //Output array to console.
    cout<<ptr[i]<<" , ";
    delete[] ptr;
    //deallocating all the memory created by new operator

}

```

Output :

```

Enter size of Array : 5
Enter any number : 78
Enter any number : 45
Enter any number : 12
Enter any number : 89
Enter any number : 56

```

```

78, 45, 12, 89, 56,

```

UNIT-2

Stack & Queue

13. Stack

A stack is an Abstract Data Type (ADT), commonly used in most programming languages. It is named stack as it behaves like a real-world stack, for example – a deck of cards or a pile of plates, etc.

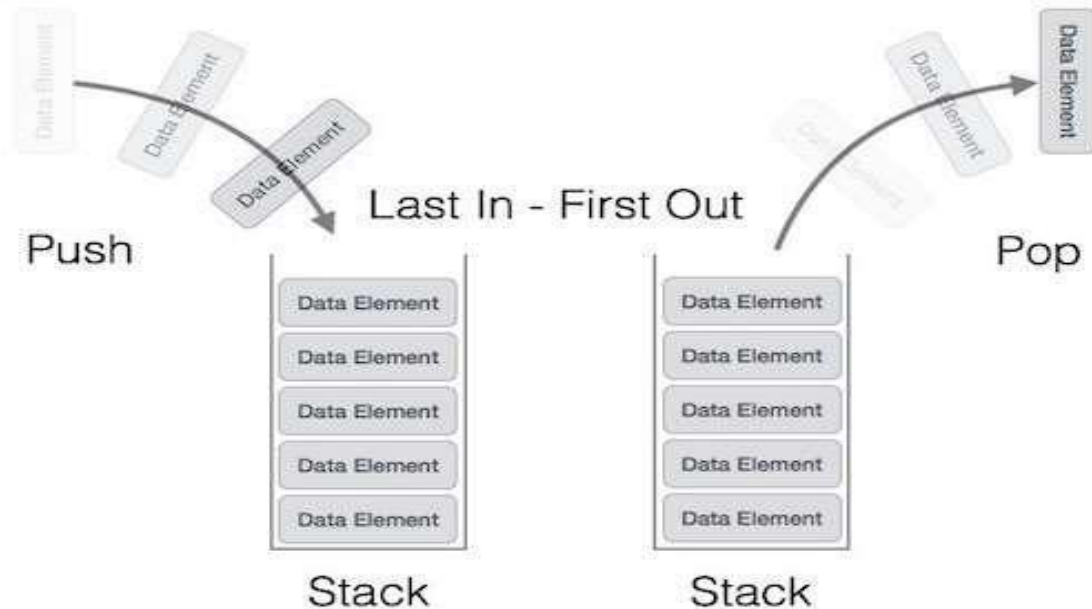


A real-world stack allows operations at one end only. For example, we can place or remove a card or plate from the top of the stack only. Likewise, Stack ADT allows all data operations at one end only. At any given time, we can only access the top element of a stack.

This feature makes it LIFO data structure. LIFO stands for Last-in-first-out. Here, the element which is placed (inserted or added) last, is accessed first. In stack terminology, insertion operation is called **PUSH** operation and removal operation is called **POP** operation.

Stack Representation

The following diagram depicts a stack and its operations –



A stack can be implemented by means of Array, Structure, Pointer, and Linked List. Stack can either be a fixed size one or it may have a sense of dynamic resizing. Here, we are going to implement stack using arrays, which makes it a fixed size stack implementation.

Basic Operations

Stack operations may involve initializing the stack, using it and then de-initializing it. Apart from these basic stuffs, a stack is used for the following two primary operations –

- **push()** – Pushing (storing) an element on the stack.
- **pop()** – Removing (accessing) an element from the stack.

When data is PUSHed onto stack.

To use a stack efficiently, we need to check the status of stack as well. For the same purpose, the following functionality is added to stacks –

- **peek()** – get the top data element of the stack, without removing it.
- **isFull()** – check if stack is full.
- **isEmpty()** – check if stack is empty.

At all times, we maintain a pointer to the last PUSHed data on the stack. As this pointer always represents the top of the stack, hence named **top**. The **top** pointer provides top value of the stack without actually removing it.

First we should learn about procedures to support stack functions –

peek()

Algorithm of peek() function –

```
begin procedure peek  
  
    return stack[top]  
  
end procedure
```

Implementation of peek() function in C programming language –

```
int peek() {  
    return stack[top];  
}
```

isfull()

Algorithm of isfull() function –

```
begin procedure isfull

    if top equals to MAXSIZE
        return true
    else
        return
    false endif

end procedure
```

Implementation of isfull() function in C programming language –

```
bool isfull() {
    if(top == MAXSIZE)
        return true;
    else
        return false;
}
```

isempty()

Algorithm of isempty() function –

```
begin procedure isempty

    if top less than 1
        return true
    else
        return
    false endif

end procedure
```

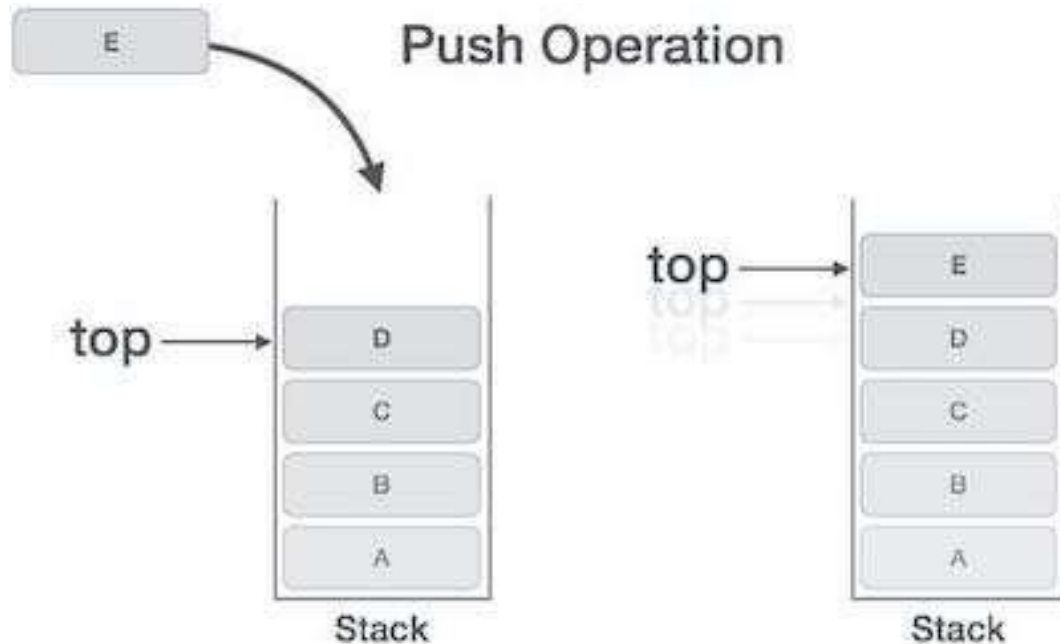
Implementation of isempty() function in C programming language is slightly different. We initialize top at -1, as the index in array starts from 0. So we check if the top is below zero or -1 to determine if the stack is empty. Here's the code –

```
bool isempty() {  
    if(top == -1)  
        return true;  
    else  
        return false;  
}
```

Push Operation

The process of putting a new data element onto stack is known as a Push Operation. Push operation involves a series of steps –

- **Step 1** – Checks if the stack is full.
- **Step 2** – If the stack is full, produces an error and exit.
- **Step 3** – If the stack is not full, increments **top** to point next empty space.
- **Step 4** – Adds data element to the stack location, where top is pointing.
- **Step 5** – Returns success.



If the linked list is used to implement the stack, then in step 3, we need to allocate space dynamically.

Algorithm for PUSH Operation

A simple algorithm for Push operation can be derived as follows –

```
begin procedure push: stack, data

    if stack is full
        return null
    endif

    top ← top + 1

    stack[top] ← data

end procedure
```

Implementation of this algorithm in C, is very easy. See the following code –

```
void push(int data) {
    if(!isFull()) {
        top = top + 1;
        stack[top] = data;
    }else {
        printf("Could not insert data, Stack is full.\n");
    }
}
```

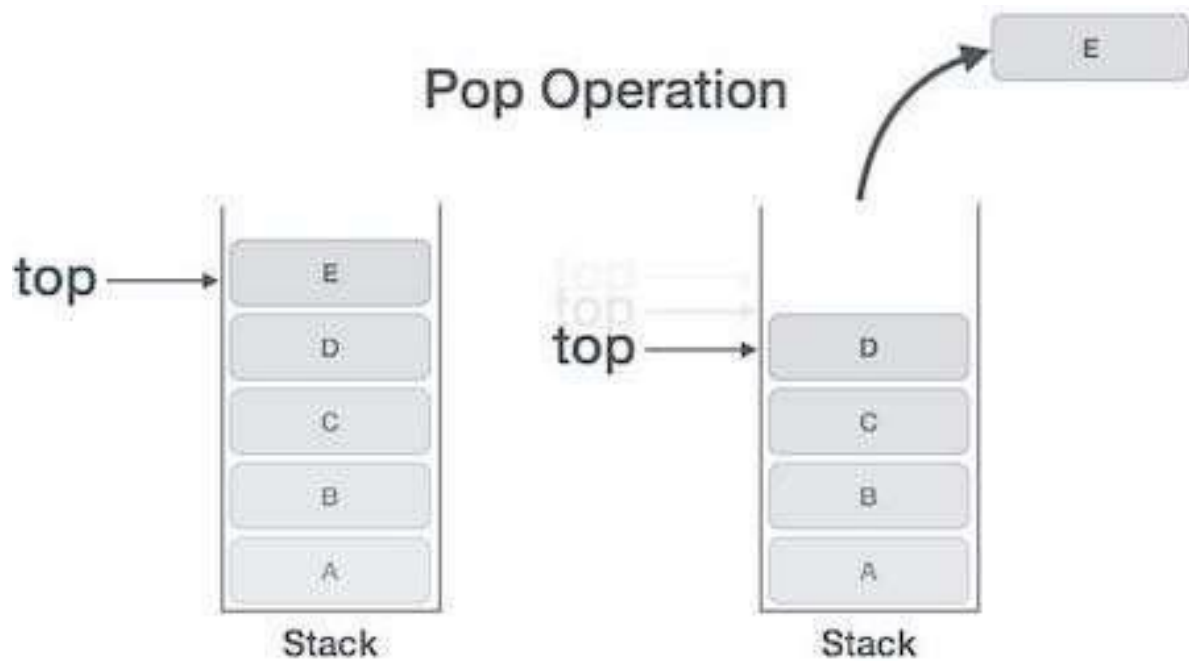
Pop Operation

Accessing the content while removing it from the stack, is known as a Pop Operation. In an array implementation of pop() operation, the data element is not actually removed, instead **top** is decremented to a lower position in the stack to point to the next value. But in linked-list implementation, pop() actually removes data element and deallocates memory space.

A Pop operation may involve the following steps –

- **Step 1** – Checks if the stack is empty.
- **Step 2** – If the stack is empty, produces an error and exit.

- **Step 3** – If the stack is not empty, accesses the data element at which **top** is pointing.
- **Step 4** – Decreases the value of top by 1.
- **Step 5** – Returns success.



Algorithm for Pop Operation

A simple algorithm for Pop operation can be derived as follows –

```
begin procedure pop: stack

    if stack is empty
        return null
    endif

    data ← stack[top]

    top ← top - 1

    return data

end procedure
```

Implementation of this algorithm in C, is as follows –

```
int pop(int data) {  
  
    if(!isempty()) {  
        data = stack[top];  
        top = top - 1;  
        return data;  
    }else {  
        printf("Could not retrieve data, Stack is empty.\n");  
    }  
}
```

For a complete stack program in C programming language, please [click here](#).

Stack Program in C

We shall see the stack implementation in C programming language here. You can try the program by clicking on the Try-it button. To learn the theory aspect of stacks, click on visit previous page.

Implementation in C

```
#include <stdio.h>  
  
int MAXSIZE = 8;  
int stack[8];  
int top = -1;  
  
int isempty() {  
  
    if(top == -1)  
        return 1;  
    else  
        return 0;  
}
```

```
int isfull() {

    if(top == MAXSIZE)
        return 1;
    else
        return 0;
}

int peek() {
    return stack[top];
}

int pop() {
    int data;

    if(!isempty()) {
        data = stack[top];
        top = top - 1;
        return data;
    }else {
        printf("Could not retrieve data, Stack is empty.\n");
    }
}

int push(int data) {

    if(!isfull()) {
        top = top + 1;
        stack[top] =
data; }else {
        printf("Could not insert data, Stack is full.\n");
    }
}
```

```
int main() {  
    // push items on to the  
    stack push(3);  
    push(5);  
    push(9);  
    push(1);  
    push(12);  
    push(15);  
  
    printf("Element at top of the stack: %d\n" ,peek());  
    printf("Elements: \n");  
  
    // print stack data  
    while(!isempty()) {  
        int data = pop();  
        printf("%d\n",data);  
    }  
  
    printf("Stack full: %s\n" , isfull()?"true":"false");  
    printf("Stack empty: %s\n" , isempty()?"true":"false");  
  
    return 0;  
}
```

If we compile and run the above program, it will produce the following result –

```
Element at top of the stack: 15  
Elements:  
15  
12  
1  
9  
5  
3  
Stack full: false  
  
Stack empty: true
```

15. Queue

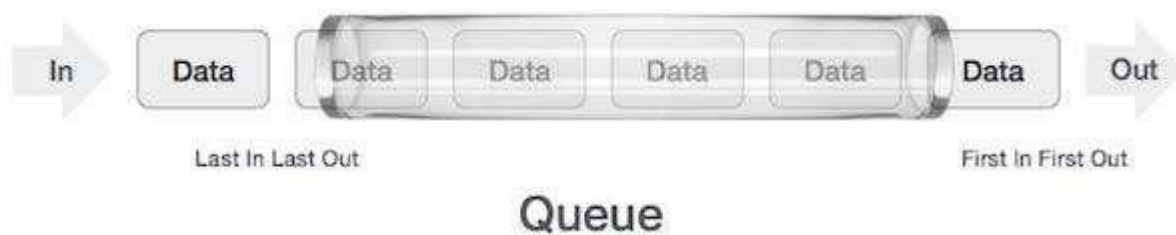
Queue is an abstract data structure, somewhat similar to Stacks. Unlike stacks, a queue is open at both its ends. One end is always used to insert data (enqueue) and the other is used to remove data (dequeue). Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first.



A real-world example of queue can be a single-lane one-way road, where the vehicle enters first, exits first. More real-world examples can be seen as queues at the ticket windows and bus-stops.

Queue Representation

As we now understand that in queue, we access both ends for different reasons. The following diagram given below tries to explain queue representation as data structure –



As in stacks, a queue can also be implemented using Arrays, Linked-lists, Pointers and Structures. For the sake of simplicity, we shall implement queues using one-dimensional array.

Basic Operations

Queue operations may involve initializing or defining the queue, utilizing it, and then completely erasing it from the memory. Here we shall try to understand the basic operations associated with queues –

- **enqueue()** – add (store) an item to the queue.
- **dequeue()** – remove (access) an item from the queue.

Few more functions are required to make the above-mentioned queue operation efficient. These are –

- **peek()** – Gets the element at the front of the queue without removing it.
- **isfull()** – Checks if the queue is full.
- **isempty()** – Checks if the queue is empty.

In queue, we always dequeue (or access) data, pointed by **front** pointer and while enqueueing (or storing) data in the queue we take help of **rear** pointer.

Let's first learn about supportive functions of a queue –

peek()

This function helps to see the data at the **front** of the queue. The algorithm of peek() function is as follows –

```
begin procedure peek  
  
    return queue[front]  
  
end procedure
```

Implementation of peek() function in C programming language –

```
int peek() {  
    return queue[front];  
}
```

isfull()

As we are using single dimension array to implement queue, we just check for the rear pointer to reach at MAXSIZE to determine that the queue is full. In case we maintain the queue in a circular linked-list, the algorithm will differ. Algorithm of isfull() function –

```
begin procedure isfull  
  
    if rear equals to MAXSIZE  
        return true  
    else
```

```
return false
endif

end procedure
```

Implementation of isfull() function in C programming language –

```
bool isfull() {
    if(rear == MAXSIZE - 1)
        return true;
    else
        return false;
}
```

isempty()

Algorithm of isempty() function –

```
begin procedure isempty

    if front is less than MIN OR front is greater than rear
        return true
    else
        return
    false endif

end procedure
```

If the value of **front** is less than MIN or 0, it tells that the queue is not yet initialized, hence empty.

Here's the C programming code –

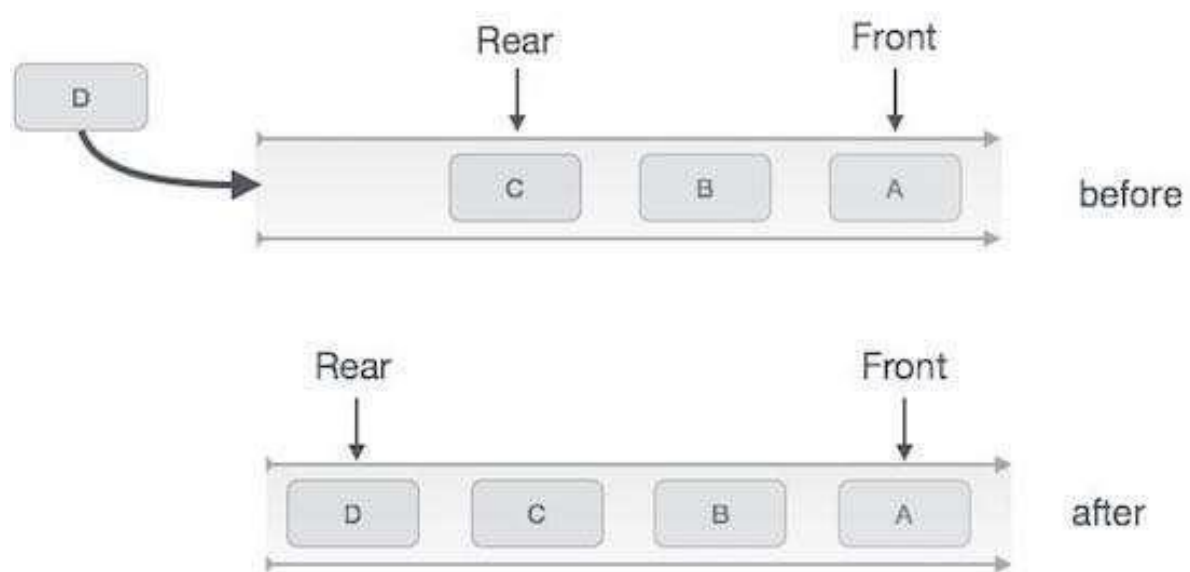
```
bool isempty() {
    if(front < 0 || front > rear)
        return true;
    else
        return false;
}
```

Enqueue Operation

Queues maintain two data pointers, **front** and **rear**. Therefore, its operations are comparatively difficult to implement than that of stacks.

The following steps should be taken to enqueue (insert) data into a queue –

- **Step 1** – Check if the queue is full.
- **Step 2** – If the queue is full, produce overflow error and exit.
- **Step 3** – If the queue is not full, increment **rear** pointer to point the next empty space.
- **Step 4** – Add data element to the queue location, where the rear is pointing.
- **Step 5** – Return success.



Queue Enqueue

Sometimes, we also check to see if a queue is initialized or not, to handle any unforeseen situations.

Algorithm for enqueue Operation

```
procedure enqueue(data)
    if queue is full
        return
    overflow endif

    rear ← rear + 1
    queue[rear] ← data
    return true

end procedure
```

Implementation of enqueue() in C programming language –

```
int enqueue(int data)
    if(isfull())
        return 0;

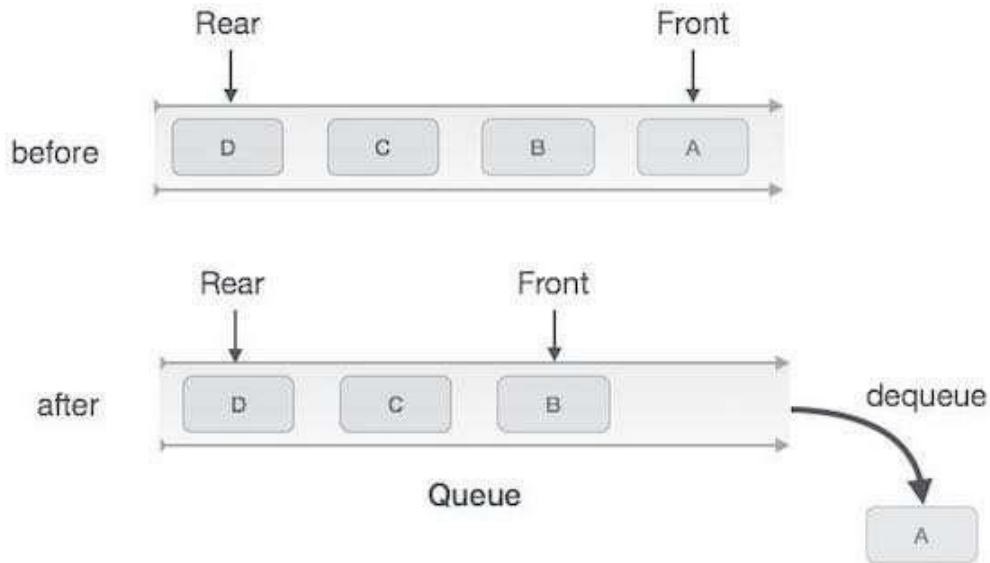
    rear = rear + 1;
    queue[rear] = data;

    return 1;
end procedure
```

Dequeue Operation

Accessing data from the queue is a process of two tasks – access the data where **front** is pointing and remove the data after access. The following steps are taken to perform **dequeue** operation –

- **Step 1** – Check if the queue is empty.
- **Step 2** – If the queue is empty, produce underflow error and exit.
- **Step 3** – If the queue is not empty, access the data where **front** is pointing.
- **Step 4** – Increment **front** pointer to point to the next available data element.
- **Step 5** – Return success.



Queue Dequeue

Algorithm for dequeue Operation

```
procedure dequeue if
    queue is empty
        return
    underflow end if

    data = queue[front]
    front ← front + 1

    return true
end procedure
```

Implementation of dequeue() in C programming language –

```
int dequeue() {
    if(isempty())
        return 0;
    int data = queue[front];
    front = front + 1;

    return data;
}
```

Queue Program in C

We shall see the stack implementation in C programming language here. You can try the program by clicking on the Try-it button. To learn the theory aspect of stacks, click on visit previous page.

Implementation in C

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>

#define MAX 6

int intArray[MAX];
int front = 0;
int rear = -1;
int itemCount = 0;

int peek(){
    return intArray[front];
}

bool isEmpty(){
    return itemCount == 0;
}

bool isFull(){
    return itemCount == MAX;
}

int size(){
    return itemCount;
}

void insert(int data){

    if(!isFull()){
```

```
        if(rear == MAX-
            1){ rear = -1;
        }

        intArray[++rear] = data;
        itemCount++;
    }
}

int removeData(){
    int data = intArray[front++];

    if(front == MAX){
        front = 0;
    }
    itemCount--;
    return data;
}

int main() {
    /* insert 5 items */
    insert(3);
    insert(5);
    insert(9);
    insert(1);
    insert(12);

    // front : 0
    // rear  : 4
    // -----
    // index : 0 1 2 3 4
    // -----
    // queue : 3 5 9 1
    12 insert(15);

    // front : 0
    // rear  : 5
```

```
// -----
// index : 0 1 2 3 4 5
// -----
// queue : 3 5 9 1 12 15

if(isFull()){
    printf("Queue is full!\n");
}

// remove one item
int num = removeData();

printf("Element removed: %d\n",num);
// front : 1
// rear  : 5
// -----
// index : 1 2 3 4 5
// -----
// queue : 5 9 1 12 15

// insert      more
items insert(16);

// front : 1
// rear  : -1
// -----
// index : 0 1 2 3 4 5 //
// -----
// queue : 16 5 9 1 12 15

// As queue is full, elements will not be
inserted. insert(17);
insert(18);

// -----
// index : 0 1 2 3 4 5 //
// -----
```

```
// queue : 16 5 9 1 12 15 printf("Element
at front: %d\n",peek());

printf("-----
\n"); printf("index : 5 4 3 2 1
0\n"); printf("-----
---\n"); printf("Queue: ");

while(!isEmpty()){
    int n = removeData();
    printf("%d ",n);
}
}
```

If we compile and run the above program, it will produce the following result –

```
Queue is full!
Element removed: 3
Element at front: 5
-----
index : 5 4 3 2 1 0
-----
Queue: 5 9 1 12 15 16
```

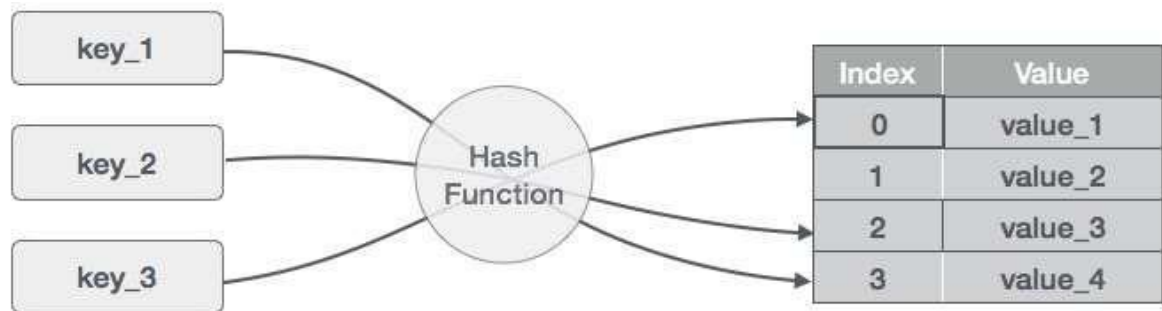
Hash Table

Hash Table is a data structure which stores data in an associative manner. In a hash table, data is stored in an array format, where each data value has its own unique index value. Access of data becomes very fast if we know the index of the desired data.

Thus, it becomes a data structure in which insertion and search operations are very fast irrespective of the size of the data. Hash Table uses an array as a storage medium and uses hash technique to generate an index where an element is to be inserted or is to be located from.

Hashing

Hashing is a technique to convert a range of key values into a range of indexes of an array. We're going to use modulo operator to get a range of key values. Consider an example of hash table of size 20, and the following items are to be stored. Item are in the (key,value) format.



- (1,20)
- (2,70)
- (42,80)
- (4,25)
- (12,44)
- (14,32)
- (17,11)
- (13,78)
- (37,98)

Sr. No.	Key	Hash	Array Index
1	1	$1 \% 20 = 1$	1
2	2	$2 \% 20 = 2$	2
3	42	$42 \% 20 = 2$	2
4	4	$4 \% 20 = 4$	4
5	12	$12 \% 20 = 12$	12
6	14	$14 \% 20 = 14$	14
7	17	$17 \% 20 = 17$	17
8	13	$13 \% 20 = 13$	13
9	37	$37 \% 20 = 17$	17

Linear Probing

As we can see, it may happen that the hashing technique is used to create an already used index of the array. In such a case, we can search the next empty location in the array by looking into the next cell until we find an empty cell. This technique is called linear probing.

Sr. No.	Key	Hash	Array Index	After Linear Probing, Array Index
1	1	$1 \% 20 = 1$	1	1
2	2	$2 \% 20 = 2$	2	2
3	42	$42 \% 20 = 2$	2	3
4	4	$4 \% 20 = 4$	4	4
5	12	$12 \% 20 = 12$	12	12
6	14	$14 \% 20 = 14$	14	14

7	17	$17 \% 20 = 17$	17	17
8	13	$13 \% 20 = 13$	13	13
9	37	$37 \% 20 = 17$	17	18

Basic Operations

Following are the basic primary operations of a hash table.

- **Search** – Searches an element in a hash table.
- **Insert** – inserts an element in a hash table.
- **Delete** – Deletes an element from a hash table.

Data Item

Define a data item having some data and key, based on which the search is to be conducted in a hash table.

```
struct DataItem {  
    int data;  
    int key;  
};
```

Hash Method

Define a hashing method to compute the hash code of the key of the data item.

```
int hashCode(int key){  
    return key % SIZE;  
}
```

Search Operation

Whenever an element is to be searched, compute the hash code of the key passed and locate the element using that hash code as index in the array. Use linear probing to get the element ahead if the element is not found at the computed hash code.

```
struct DataItem *search(int key){
    //get the hash
    int hashIndex = hashCode(key);

    //move in array until an empty
    while(hashArray[hashIndex] != NULL){

        if(hashArray[hashIndex]->key == key)
            return hashArray[hashIndex];

        //go to next cell
        ++hashIndex;

        //wrap around the table
        hashIndex %= SIZE;
    }

    return NULL;
}
```

Insert Operation

Whenever an element is to be inserted, compute the hash code of the key passed and locate the index using that hash code as an index in the array. Use linear probing for empty location, if an element is found at the computed hash code.

```
void insert(int key,int data){
    struct DataItem *item = (struct DataItem*) malloc(sizeof(struct
    DataItem)); item->data = data;
    item->key = key;

    //get the hash
    int hashIndex = hashCode(key);

    //move in array until an empty or deleted cell while(hashArray[hashIndex]
    != NULL && hashArray[hashIndex]->key != -1){
        //go to next cell
        ++hashIndex;
    }
```

```
        //wrap around the table
        hashIndex %= SIZE;
    }

    hashArray[hashIndex] = item;
}
```

Delete Operation

Whenever an element is to be deleted, compute the hash code of the key passed and locate the index using that hash code as an index in the array. Use linear probing to get the element ahead if an element is not found at the computed hash code. When found, store a dummy item there to keep the performance of the hash table intact.

```
struct DataItem* delete(struct DataItem* item){
    int key = item->key;

    //get the hash
    int hashIndex = hashCode(key);

    //move in array until an empty
    while(hashArray[hashIndex] !=NULL){

        if(hashArray[hashIndex]->key == key){
            struct DataItem* temp = hashArray[hashIndex];

            //assign a dummy item at deleted position
            hashArray[hashIndex] = dummyItem;
            return temp;
        }
        //go to next cell
        ++hashIndex;

        //wrap around the table
        hashIndex %= SIZE;
    }
    return NULL;
}
```

Hash Table Program in C

Hash Table is a data structure which stores data in an associative manner. In hash table, the data is stored in an array format where each data value has its own unique index value. Access of data becomes very fast, if we know the index of the desired data.

Implementation in C

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>

#define SIZE 20

struct DataItem {
    int data;
    int key;
};

struct DataItem* hashArray[SIZE];
struct DataItem* dummyItem;
struct DataItem* item;

int hashCode(int key){
    return key % SIZE;
}

struct DataItem *search(int key){
    //get the hash
    int hashIndex = hashCode(key);

    //move in array until an empty
    while(hashArray[hashIndex] != NULL){

        if(hashArray[hashIndex]->key == key)
            return hashArray[hashIndex];
```

```
        //go to next cell
        ++hashIndex;

        //wrap around the table
        hashIndex %= SIZE;
    }

    return NULL;
}

void insert(int key,int data){

    struct DataItem *item = (struct DataItem*) malloc(sizeof(struct
    DataItem)); item->data = data;
    item->key = key;

    //get the hash
    int hashIndex = hashCode(key);

    //move in array until an empty or deleted cell while(hashArray[hashIndex]
    != NULL && hashArray[hashIndex]->key != -1){

        //go to next cell
        ++hashIndex;

        //wrap around the table
        hashIndex %= SIZE;
    }

    hashArray[hashIndex] = item;
}

struct DataItem* delete(struct DataItem* item){
    int key = item->key;

    //get the hash
```

```
int hashIndex = hashCode(key);

//move in array until an empty
while(hashArray[hashIndex] != NULL){

    if(hashArray[hashIndex]->key == key){
        struct DataItem* temp = hashArray[hashIndex];

        //assign a dummy item at deleted position
        hashArray[hashIndex] = dummyItem;
        return temp;
    }

    //go to next cell
    ++hashIndex;

    //wrap around the table
    hashIndex %= SIZE;
}

return NULL;
}

void display(){
    int i = 0;

    for(i = 0; i<SIZE; i++) {

        if(hashArray[i] != NULL)
            printf(" (%d,%d)",hashArray[i]->key,hashArray[i]->data);
        else
            printf(" ~~ ");
    }

    printf("\n");
}
```

```
int main(){
    dummyItem = (struct DataItem*) malloc(sizeof(struct
DataItem)); dummyItem->data = -1;
    dummyItem->key = -1;

    insert(1, 20);
    insert(2, 70);
    insert(42, 80);
    insert(4, 25);
    insert(12, 44);
    insert(14, 32);
    insert(17, 11);
    insert(13, 78);
    insert(37, 97);

    display();

    item = search(37);

    if(item != NULL){
        printf("Element found: %d\n", item-
>data); }else {
        printf("Element not found\n");
    }

    delete(item);

    item = search(37);

    if(item != NULL){
        printf("Element found: %d\n", item-
>data); }else {
        printf("Element not found\n");
    }
}
```

If we compile and run the above program, it will produce the following result –

```
~~ (1,20) (2,70) (42,80) (4,25) ~~ ~~ ~~ ~~ ~~ ~~ (12,44)
(13,78) (14,32) ~~ ~~ (17,11) (37,97) ~~
Element found: 97
Element not found
```


Binary Trees

by Nick Parlante

This article introduces the basic concepts of binary trees, and then works through a series of practice problems with solution code in C/C++ and Java. Binary trees have an elegant recursive pointer structure, so they are a good way to learn recursive pointer algorithms.

Contents

[Section 1](#). Binary Tree Structure -- a quick introduction to binary trees and the code that operates on them

[Section 2](#). Binary Tree Problems -- practice problems in increasing order of difficulty

[Section 3](#). C Solutions -- solution code to the problems for C and C++ programmers

[Section 4](#). Java versions -- how binary trees work in Java, with solution code

Stanford CS Education Library -- #110

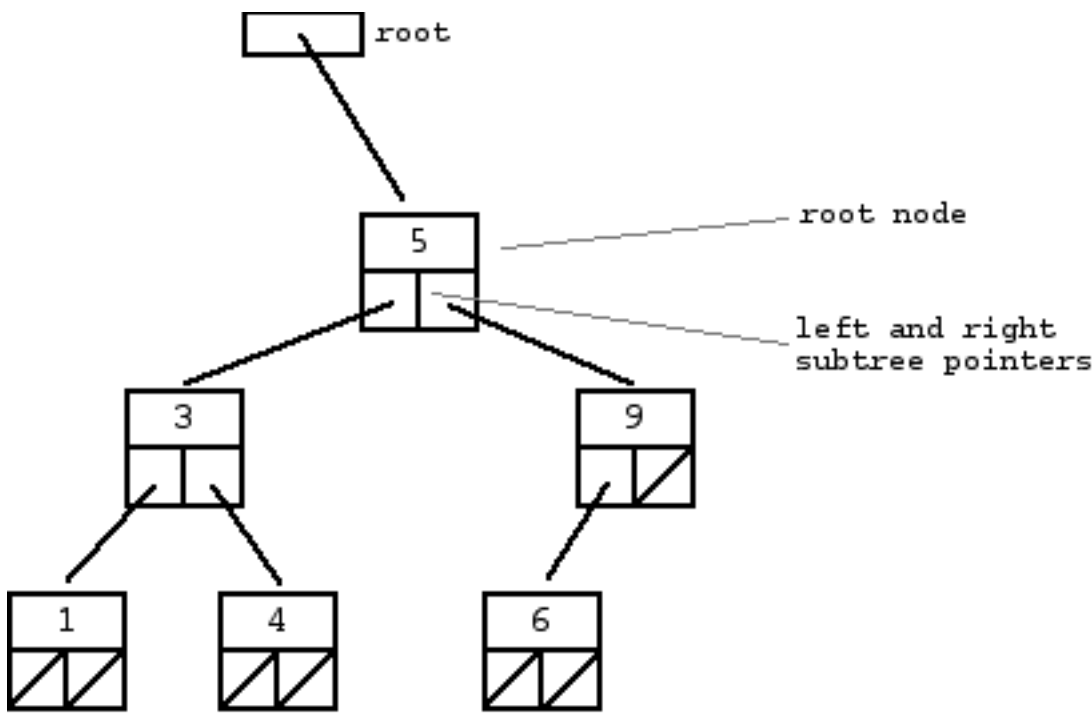
This is article #110 in the Stanford CS Education Library. This and other free CS materials are available at the library (<http://cslibrary.stanford.edu/>). That people seeking education should have the opportunity to find it. This article may be used, reproduced, excerpted, or sold so long as this paragraph is clearly reproduced. Copyright 2000-2001, Nick Parlante, nick.parlante@cs.stanford.edu.

Related CSLibrary Articles

- [Linked List Problems \(http://cslibrary.stanford.edu/105/\)](http://cslibrary.stanford.edu/105/) -- a large collection of linked list problems using various pointer techniques (while this binary tree article concentrates on recursion)
- [Pointer and Memory \(http://cslibrary.stanford.edu/102/\)](http://cslibrary.stanford.edu/102/) -- basic concepts of pointers and memory
- [The Great Tree-List Problem \(http://cslibrary.stanford.edu/109/\)](http://cslibrary.stanford.edu/109/) -- a great pointer recursion problem that uses both trees and lists

Section 1 -- Introduction To Binary Trees

A binary tree is made of nodes, where each node contains a "left" pointer, a "right" pointer, and a data element. The "root" pointer points to the topmost node in the tree. The left and right pointers recursively point to smaller "subtrees" on either side. A null pointer represents a binary tree with no elements -- the empty tree. The formal recursive definition is: a **binary tree** is either empty (represented by a null pointer), or is made of a single node, where the left and right pointers (recursive definition ahead) each point to a **binary tree**.



A "binary search tree" (BST) or "ordered binary tree" is a type of binary tree where the nodes are arranged in order: for each node, all elements in its left subtree are less-or-equal to the node (\leq), and all the elements in its right subtree are greater than the node ($>$). The tree shown above is a binary search tree -- the "root" node is a 5, and its left subtree nodes (1, 3, 4) are ≤ 5 , and its right subtree nodes (6, 9) are > 5 . Recursively, each of the subtrees must also obey the binary search tree constraint: in the (1, 3, 4) subtree, the 3 is the root, the $1 \leq 3$ and $4 > 3$. Watch out for the exact wording in the problems -- a "binary search tree" is different from a "binary tree".

The nodes at the bottom edge of the tree have empty subtrees and are called "leaf" nodes (1, 4, 6) while the others are "internal" nodes (3, 5, 9).

Binary Search Tree Niche

Basically, binary search trees are fast at insert and lookup. The next section presents the code for these two algorithms. On average, a binary search tree algorithm can locate a node in an N node tree in order $\lg(N)$ time (log base 2). Therefore, binary search trees are good for "dictionary" problems where the code inserts and looks up information indexed by some key. The $\lg(N)$ behavior is the average case -- it's possible for a particular tree to be much slower depending on its shape.

Strategy

Some of the problems in this article use plain binary trees, and some use binary search trees. In any case, the problems concentrate on the combination of pointers and recursion. (See the articles linked above for pointer articles that do not emphasize recursion.)

For each problem, there are two things to understand...

- The node/pointer structure that makes up the tree and the code that manipulates it
- The algorithm, typically recursive, that iterates over the tree

When thinking about a binary tree problem, it's often a good idea to draw a few little trees to think about the various cases.

Typical Binary Tree Code in C/C++

As an introduction, we'll look at the code for the two most basic binary search tree operations -- `lookup()` and `insert()`. The code here works for C or C++. Java programmers can read the discussion here, and then look at the Java versions in [Section 4](#).

In C or C++, the binary tree is built with a node type like this...

```
struct node {
    int data;
    struct node* left;
    struct node* right;
}
```

Lookup()

Given a binary search tree and a "target" value, search the tree to see if it contains the target. The basic pattern of the `lookup()` code occurs in many recursive tree algorithms: deal with the base case where the tree is empty, deal with the current node, and then use recursion to deal with the subtrees. If the tree is a binary search tree, there is often some sort of less-than test on the node to decide if the recursion should go left or right.

```
/*
Given a binary tree, return true if a node
with the target data is found in the tree. Recurs
down the tree, chooses the left or right
branch by comparing the target to each node.
*/
static int lookup(struct node* node, int target) {
    // 1. Base case == empty tree
    // in that case, the target is not found so return false
    if (node == NULL) {
        return(false);
    }
    else {
        // 2. see if found here
        if (target == node->data) return(true);
        else {
            // 3. otherwise recur down the correct subtree
            if (target < node->data) return(lookup(node->left, target));
            else return(lookup(node->right, target));
        }
    }
}
```

The `lookup()` algorithm could be written as a while-loop that iterates down the tree. Our version uses recursion to help prepare you for the problems below that require recursion.

Pointer Changing Code

There is a common problem with pointer intensive code: what if a function needs to change one of the pointer parameters passed to it? For example, the `insert()` function below may want to change the root pointer. In C and C++, one solution uses pointers-to-pointers (aka "reference parameters"). That's a fine technique, but here we will use the simpler technique that a function that wishes to change a pointer passed to it will **return** the new value of the pointer to the caller. The caller is responsible for using the new value. Suppose we have a `change()` function

that may change the the root, then a call to `change()` will look like this...

```
// suppose the variable "root" points to the tree
root = change(root);
```

We take the value returned by `change()`, and use it as the new value for `root`. This construct is a little awkward, but it avoids using reference parameters which confuse some C and C++ programmers, and Java does not have reference parameters at all. This allows us to focus on the recursion instead of the pointer mechanics. (For lots of problems that use reference parameters, see CSLibrary #105, Linked List Problems, <http://cslibrary.stanford.edu/105/>).

Insert()

`Insert()` -- given a binary search tree and a number, insert a new node with the given number into the tree in the correct place. The `insert()` code is similar to `lookup()`, but with the complication that it modifies the tree structure. As described above, `insert()` returns the new tree pointer to use to its caller. Calling `insert()` with the number 5 on this tree...

```

      2
     / \
    1   10
```

returns the tree...

```

      2
     / \
    1   10
     /
    5
```

The solution shown here introduces a `newNode()` helper function that builds a single node. The base-case/recursion structure is similar to the structure in `lookup()` -- each call checks for the NULL case, looks at the node at hand, and then recurs down the left or right subtree if needed.

```
/*
 * Helper function that allocates a new node
 * with the given data and NULL left and right
 * pointers.
 */
struct node* NewNode(int data) {
    struct node* node = new(struct node);    // "new" is like "malloc"
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return(node);
}

/*
 * Give a binary search tree and a number, inserts a new node
 * with the given number in the correct place in the tree.
 * Returns the new root pointer which the caller should
 * then use (the standard trick to avoid using reference
 * parameters).
 */
struct node* insert(struct node* node, int data) {
```

```

// 1. If the tree is empty, return a new, single node
if (node == NULL) {
    return(newNode(data));
}
else {
    // 2. Otherwise, recur down the tree
    if (data <= node->data) node->left = insert(node->left, data);
    else node->right = insert(node->right, data);

    return(node); // return the (unchanged) node pointer
}
}

```

The shape of a binary tree depends very much on the order that the nodes are inserted. In particular, if the nodes are inserted in increasing order (1, 2, 3, 4), the tree nodes just grow to the right leading to a linked list shape where all the left pointers are NULL. A similar thing happens if the nodes are inserted in decreasing order (4, 3, 2, 1). The linked list shape defeats the $\lg(N)$ performance. We will not address that issue here, instead focusing on pointers and recursion.

Section 2 -- Binary Tree Problems

Here are 14 binary tree problems in increasing order of difficulty. Some of the problems operate on binary search trees (aka "ordered binary trees") while others work on plain binary trees with no special ordering. The next section, [Section 3](#), shows the solution code in C/C++. [Section 4](#) gives the background and solution code in Java. The basic structure and recursion of the solution code is the same in both languages -- the differences are superficial.

Reading about a data structure is a fine introduction, but at some point the only way to learn is to actually try to solve some problems starting with a blank sheet of paper. To get the most out of these problems, you should at least attempt to solve them before looking at the solution. Even if your solution is not quite right, you will be building up the right skills. With any pointer-based code, it's a good idea to make memory drawings of a few simple cases to see how the algorithm should work.

1. build123()

This is a very basic problem with a little pointer manipulation. (You can skip this problem if you are already comfortable with pointers.) Write code that builds the following little 1-2-3 binary search tree...

```

      2
     / \
    1   3

```

Write the code in three different ways...

- a: by calling newNode() three times, and using three pointer variables
- b: by calling newNode() three times, and using only one pointer variable
- c: by calling insert() three times passing it the root pointer to build up the tree

(In Java, write a build123() method that operates on the receiver to change it to be the 1-2-3 tree with the given coding constraints. See [Section 4](#).)

```
struct node* build123() {
```

2. size()

This problem demonstrates simple binary tree traversal. Given a binary tree, count the number of nodes in the tree.

```
int size(struct node* node) {
```

3. maxDepth()

Given a binary tree, compute its "maxDepth" -- the number of nodes along the longest path from the root node down to the farthest leaf node. The maxDepth of the empty tree is 0, the maxDepth of the tree on the first page is 3.

```
int maxDepth(struct node* node) {
```

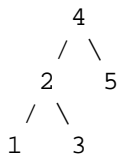
4. minValue()

Given a non-empty binary search tree (an ordered binary tree), return the minimum data value found in that tree. Note that it is not necessary to search the entire tree. A maxVal() function is structurally very similar to this function. This can be solved with recursion or with a simple while loop.

```
int minValue(struct node* node) {
```

5. printTree()

Given a binary search tree (aka an "ordered binary tree"), iterate over the nodes to print them out in increasing order. So the tree...



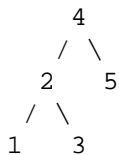
Produces the output "1 2 3 4 5". This is known as an "inorder" traversal of the tree.

Hint: For each node, the strategy is: recur left, print the node data, recur right.

```
void printTree(struct node* node) {
```

6. printPostorder()

Given a binary tree, print out the nodes of the tree according to a bottom-up "postorder" traversal -- both subtrees of a node are printed out completely before the node itself is printed, and each left subtree is printed before the right subtree. So the tree...

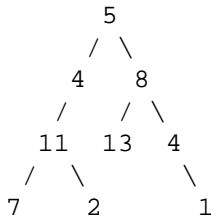


Produces the output "1 3 2 5 4". The description is complex, but the code is simple. This is the sort of bottom-up traversal that would be used, for example, to evaluate an expression tree where a node is an operation like '+' and its subtrees are, recursively, the two subexpressions for the '+'.

```
void printPostorder(struct node* node) {
```

7. hasPathSum()

We'll define a "root-to-leaf path" to be a sequence of nodes in a tree starting with the root node and proceeding downward to a leaf (a node with no children). We'll say that an empty tree contains no root-to-leaf paths. So for example, the following tree has exactly four root-to-leaf paths:



Root-to-leaf paths:

```

path 1: 5 4 11 7
path 2: 5 4 11 2
path 3: 5 8 13
path 4: 5 8 4 1

```

For this problem, we will be concerned with the sum of the values of such a path -- for example, the sum of the values on the 5-4-11-7 path is $5 + 4 + 11 + 7 = 27$.

Given a binary tree and a sum, return true if the tree has a root-to-leaf path such that adding up all the values along the path equals the given sum. Return false if no such path can be found. (Thanks to Owen Astrachan for suggesting this problem.)

```
int hasPathSum(struct node* node, int sum) {
```

8. printPaths()

Given a binary tree, print out all of its root-to-leaf paths as defined above. This problem is a little harder than it looks, since the "path so far" needs to be communicated between the recursive calls. **Hint:** In C, C++, and Java, probably the best solution is to create a recursive helper function `printPathsRecur(node, int path[], int pathLen)`, where the path array communicates the sequence of nodes that led up to the current call. Alternately, the problem may be solved bottom-up, with each node returning its list of paths. This strategy works quite nicely in Lisp, since it can exploit the built in list and mapping primitives. (Thanks to Matthias Felleisen for suggesting this problem.)

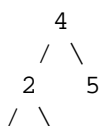
Given a binary tree, print out all of its root-to-leaf paths, one per line.

```
void printPaths(struct node* node) {
```

9. mirror()

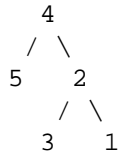
Change a tree so that the roles of the left and right pointers are swapped at every node.

So the tree...



1 3

is changed to...



The solution is short, but very recursive. As it happens, this can be accomplished without changing the root node pointer, so the return-the-new-root construct is not necessary. Alternately, if you do not want to change the tree nodes, you may construct and return a new mirror tree based on the original tree.

```
void mirror(struct node* node) {
```

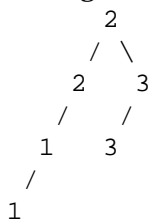
10. doubleTree()

For each node in a binary search tree, create a new duplicate node, and insert the duplicate as the left child of the original node. The resulting tree should still be a binary search tree.

So the tree...



is changed to...



As with the previous problem, this can be accomplished without changing the root node pointer.

```
void doubleTree(struct node* node) {
```

11. sameTree()

Given two binary trees, return true if they are structurally identical -- they are made of nodes with the same values arranged in the same way. (Thanks to Julie Zelenski for suggesting this problem.)

```
int sameTree(struct node* a, struct node* b) {
```

12. countTrees()

This is not a binary tree programming problem in the ordinary sense -- it's more of a math/combinatorics recursion problem that happens to use binary trees. (Thanks to Jerry Cain for suggesting this problem.)

Suppose you are building an N node binary search tree with the values 1..N. How many structurally different binary search trees are there that store those values? Write a recursive function that, given the number of distinct values, computes the number of structurally unique binary search trees that store those values. For example,

countTrees(4) should return 14, since there are 14 structurally unique binary search trees that store 1, 2, 3, and 4. The base case is easy, and the recursion is short but dense. Your code should not construct any actual trees; it's just a counting problem.

```
int countTrees(int numKeys) {
```

Binary Search Tree Checking (for problems 13 and 14)

This background is used by the next two problems: Given a plain binary tree, examine the tree to determine if it meets the requirement to be a binary search tree. To be a binary search tree, for every node, all of the nodes in its left tree must be \leq the node, and all of the nodes in its right subtree must be $>$ the node. Consider the following four examples...

a. 5 -> TRUE
 / \
 2 7

b. 5 -> FALSE, because the 6 is not ok to the left of the 5
 / \
 6 7

c. 5 -> TRUE
 / \
 2 7
 /
 1

d. 5 -> FALSE, the 6 is ok with the 2, but the 6 is not ok with the 5
 / \
 2 7
 / \
 1 6

For the first two cases, the right answer can be seen just by comparing each node to the two nodes immediately below it. However, the fourth case shows how checking the BST quality may depend on nodes which are several layers apart -- the 5 and the 6 in that case.

13 isBST() -- version 1

Suppose you have helper functions minValue() and maxValue() that return the min or max int value from a non-empty tree (see problem 3 above). Write an isBST() function that returns true if a tree is a binary search tree and false otherwise. Use the helper functions, and don't forget to check every node in the tree. It's ok if your solution is not very efficient. (Thanks to Owen Astrachan for the idea of having this problem, and comparing it to problem 14)

Returns true if a binary tree is a binary search tree.

```
int isBST(struct node* node) {
```

14. isBST() -- version 2

Version 1 above runs slowly since it traverses over some parts of the tree many times. A better solution looks at each node only once. The trick is to write a utility helper function `isBSTRecur(struct node* node, int min, int max)` that traverses down the tree keeping track of the narrowing min and max allowed values as it goes, looking at each node only once. The initial values for min and max should be `INT_MIN` and `INT_MAX` -- they narrow from there.

```
/*
Returns true if the given tree is a binary search tree
(efficient version).
*/
int isBST2(struct node* node) {
    return(isBSTRecur(node, INT_MIN, INT_MAX));
}

/*
Returns true if the given tree is a BST and its
values are >= min and <= max.
*/
int isBSTRecur(struct node* node, int min, int max) {
```

15. Tree-List

The Tree-List problem is one of the greatest recursive pointer problems ever devised, and it happens to use binary trees as well. CLibrary #109 <http://cslibrary.stanford.edu/109/> works through the Tree-List problem in detail and includes solution code in C and Java. The problem requires an understanding of binary trees, linked lists, recursion, and pointers. It's a great problem, but it's complex.

Section 3 -- C/C++ Solutions

Make an attempt to solve each problem before looking at the solution -- it's the best way to learn.

1. Build1230 Solution (C/C++)

```
// call newNode() three times
struct node* build123a() {
    struct node* root = newNode(2);
    struct node* lChild = newNode(1);
    struct node* rChild = newNode(3);

    root->left = lChild;
    root->right = rChild;

    return(root);
}

// call newNode() three times, and use only one local variable
struct node* build123b() {
    struct node* root = newNode(2);
    root->left = newNode(1);
    root->right = newNode(3);

    return(root);
}
```

```

/*
Build 123 by calling insert() three times.
Note that the '2' must be inserted first.
*/
struct node* build123c() {
    struct node* root = NULL;
    root = insert(root, 2);
    root = insert(root, 1);
    root = insert(root, 3);
    return(root);
}

```

2. size() Solution (C/C++)

```

/*
Compute the number of nodes in a tree.
*/
int size(struct node* node) {
    if (node==NULL) {
        return(0);
    } else {
        return(size(node->left) + 1 + size(node->right));
    }
}

```

3. maxDepth() Solution (C/C++)

```

/*
Compute the "maxDepth" of a tree -- the number of nodes along
the longest path from the root node down to the farthest leaf node.
*/
int maxDepth(struct node* node) {
    if (node==NULL) {
        return(0);
    }
    else {
        // compute the depth of each subtree
        int lDepth = maxDepth(node->left);
        int rDepth = maxDepth(node->right);

        // use the larger one
        if (lDepth > rDepth) return(lDepth+1);
        else return(rDepth+1);
    }
}

```

4. minValue() Solution (C/C++)

```

/*
Given a non-empty binary search tree,
return the minimum data value found in that tree.

```

Note that the entire tree does not need to be searched.

```

*/
int minValue(struct node* node) {
    struct node* current = node;

    // loop down to find the leftmost leaf
    while (current->left != NULL) {
        current = current->left;
    }

    return(current->data);
}

```

5. printTree() Solution (C/C++)

```

/*
Given a binary search tree, print out
its data elements in increasing
sorted order.
*/
void printTree(struct node* node) {
    if (node == NULL) return;

    printTree(node->left);
    printf("%d ", node->data);
    printTree(node->right);
}

```

6. printPostorder() Solution (C/C++)

```

/*
Given a binary tree, print its
nodes according to the "bottom-up"
postorder traversal.
*/
void printPostorder(struct node* node) {
    if (node == NULL) return;

    // first recur on both subtrees
    printTree(node->left);
    printTree(node->right);

    // then deal with the node
    printf("%d ", node->data);
}

```

7. hasPathSum() Solution (C/C++)

```

/*
Given a tree and a sum, return true if there is a path from the root
down to a leaf, such that adding up all the values along the path
equals the given sum.

```

Strategy: subtract the node value from the sum when recurring down, and check to see if the sum is 0 when you run out of tree.

```

*/
int hasPathSum(struct node* node, int sum) {
    // return true if we run out of tree and sum==0
    if (node == NULL) {
        return(sum == 0);
    }
    else {
        // otherwise check both subtrees
        int subSum = sum - node->data;
        return(hasPathSum(node->left, subSum) ||
               hasPathSum(node->right, subSum));
    }
}

```

8. printPaths() Solution (C/C++)

```

/*
Given a binary tree, print out all of its root-to-leaf
paths, one per line. Uses a recursive helper to do the work.
*/
void printPaths(struct node* node) {
    int path[1000];

    printPathsRecur(node, path, 0);
}

/*
Recursive helper function -- given a node, and an array containing
the path from the root node up to but not including this node,
print out all the root-leaf paths.
*/
void printPathsRecur(struct node* node, int path[], int pathLen) {
    if (node==NULL) return;

    // append this node to the path array
    path[pathLen] = node->data;
    pathLen++;

    // it's a leaf, so print the path that led to here
    if (node->left==NULL && node->right==NULL) {
        printArray(path, pathLen);
    }
    else {
        // otherwise try both subtrees
        printPathsRecur(node->left, path, pathLen);
        printPathsRecur(node->right, path, pathLen);
    }
}

// Utility that prints out an array on a line.
void printArray(int ints[], int len) {
    int i;
    for (i=0; i<len; i++) {

```

```

    printf("%d ", ints[i]);
}
printf("\n");
}

```

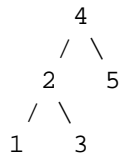
9. mirror() Solution (C/C++)

```

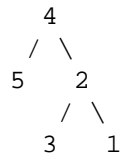
/*
Change a tree so that the roles of the
left and right pointers are swapped at every node.

```

So the tree...



is changed to...



```

*/
void mirror(struct node* node) {
    if (node==NULL) {
        return;
    }
    else {
        struct node* temp;

        // do the subtrees
        mirror(node->left);
        mirror(node->right);

        // swap the pointers in this node
        temp = node->left;
        node->left = node->right;
        node->right = temp;
    }
}

```

10. doubleTree() Solution (C/C++)

```

/*
For each node in a binary search tree,
create a new duplicate node, and insert
the duplicate as the left child of the original node.
The resulting tree should still be a binary search tree.

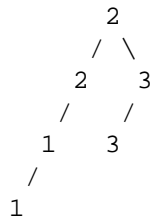
```

So the tree...



1 3

Is changed to...



```

*/
void doubleTree(struct node* node) {
    struct node* oldLeft;

    if (node==NULL) return;

    // do the subtrees
    doubleTree(node->left);
    doubleTree(node->right);

    // duplicate this node to its left
    oldLeft = node->left;
    node->left = newNode(node->data);
    node->left->left = oldLeft;
}

```

11. sameTree() Solution (C/C++)

```

/*
Given two trees, return true if they are
structurally identical.
*/
int sameTree(struct node* a, struct node* b) {
    // 1. both empty -> true
    if (a==NULL && b==NULL) return(true);

    // 2. both non-empty -> compare them
    else if (a!=NULL && b!=NULL) {
        return(
            a->data == b->data &&
            sameTree(a->left, b->left) &&
            sameTree(a->right, b->right)
        );
    }
    // 3. one empty, one not -> false
    else return(false);
}

```

12. countTrees() Solution (C/C++)

```

/*
For the key values 1...numKeys, how many structurally unique

```

binary search trees are possible that store those keys.

Strategy: consider that each value could be the root.

Recursively find the size of the left and right subtrees.

```

*/
int countTrees(int numKeys) {

    if (numKeys <=1) {
        return(1);
    }
    else {
        // there will be one value at the root, with whatever remains
        // on the left and right each forming their own subtrees.
        // Iterate through all the values that could be the root...
        int sum = 0;
        int left, right, root;

        for (root=1; root<=numKeys; root++) {
            left = countTrees(root - 1);
            right = countTrees(numKeys - root);

            // number of possible trees with this root == left*right
            sum += left*right;
        }

        return(sum);
    }
}

```

13. isBST1() Solution (C/C++)

```

/*
Returns true if a binary tree is a binary search tree.
*/
int isBST(struct node* node) {
    if (node==NULL) return(true);

    // false if the min of the left is > than us
    if (node->left!=NULL && minValue(node->left) > node->data)
        return(false);

    // false if the max of the right is <= than us
    if (node->right!=NULL && maxValue(node->right) <= node->data)
        return(false);

    // false if, recursively, the left or right is not a BST
    if (!isBST(node->left) || !isBST(node->right))
        return(false);

    // passing all that, it's a BST
    return(true);
}

```


14. isBST2() Solution (C/C++)

```

/*
Returns true if the given tree is a binary search tree
(efficient version).
*/
int isBST2(struct node* node) {
    return(isBSTUtil(node, INT_MIN, INT_MAX));
}

/*
Returns true if the given tree is a BST and its
values are >= min and <= max.
*/
int isBSTUtil(struct node* node, int min, int max) {
    if (node==NULL) return(true);

    // false if this node violates the min/max constraint
    if (node->data<min || node->data>max) return(false);

    // otherwise check the subtrees recursively,
    // tightening the min or max constraint
    return
        isBSTUtil(node->left, min, node->data) &&
        isBSTUtil(node->right, node->data+1, max)
    );
}

```

15. TreeList Solution (C/C++)

The solution code in C and Java to the great Tree-List recursion problem is in CSLibrary #109
<http://cslibrary.stanford.edu/109/>

Section 4 -- Java Binary Trees and Solutions

In Java, the key points in the recursion are exactly the same as in C or C++. In fact, I created the Java solutions by just copying the C solutions, and then making the syntactic changes. The recursion is the same, however the outer structure is slightly different.

In Java, we will have a BinaryTree object that contains a single root pointer. The root pointer points to an internal Node class that behaves just like the node struct in the C/C++ version. The Node class is private -- it is used only for internal storage inside the BinaryTree and is not exposed to clients. With this OOP structure, almost every operation has two methods: a one-line method on the BinaryTree that starts the computation, and a recursive method that works on the Node objects. For the lookup() operation, there is a BinaryTree.lookup() method that the client uses to start a lookup operation. Internal to the BinaryTree class, there is a private recursive lookup(Node) method that implements the recursion down the Node structure. This second, private recursive method is basically the same as the recursive C/C++ functions above -- it takes a Node argument and uses recursion to iterate over the pointer structure.

Java Binary Tree Structure

To get started, here are the basic definitions for the Java BinaryTree class, and the lookup() and insert() methods as examples...

```
// BinaryTree.java
public class BinaryTree {
    // Root node pointer. Will be null for an empty tree.
    private Node root;

    /*
    --Node--
    The binary tree is built using this nested node class.
    Each node stores one data element, and has left and right
    sub-tree pointer which may be null.
    The node is a "dumb" nested class -- we just use it for
    storage; it does not have any methods.
    */
    private static class Node {
        Node left;
        Node right;
        int data;

        Node(int newData) {
            left = null;
            right = null;
            data = newData;
        }
    }

    /**
    Creates an empty binary tree -- a null root pointer.
    */
    public void BinaryTree() {
        root = null;
    }

    /**
    Returns true if the given target is in the binary tree.
    Uses a recursive helper.
    */
    public boolean lookup(int data) {
        return(lookup(root, data));
    }

    /**
    Recursive lookup -- given a node, recur
    down searching for the given data.
    */
    private boolean lookup(Node node, int data) {
        if (node==null) {
            return(false);
        }

        if (data==node.data) {
            return(true);
        }
        else if (data<node.data) {

```

```

        return(lookup(node.left, data));
    }
    else {
        return(lookup(node.right, data));
    }
}

/**
 * Inserts the given data into the binary tree.
 * Uses a recursive helper.
 */
public void insert(int data) {
    root = insert(root, data);
}

/**
 * Recursive insert -- given a node pointer, recur down and
 * insert the given data into the tree. Returns the new
 * node pointer (the standard way to communicate
 * a changed pointer back to the caller).
 */
private Node insert(Node node, int data) {
    if (node==null) {
        node = new Node(data);
    }
    else {
        if (data <= node.data) {
            node.left = insert(node.left, data);
        }
        else {
            node.right = insert(node.right, data);
        }
    }

    return(node); // in any case, return the new pointer to the caller
}

```

OOP Style vs. Recursive Style

From the client point of view, the `BinaryTree` class demonstrates good OOP style -- it encapsulates the binary tree state, and the client sends messages like `lookup()` and `insert()` to operate on that state. Internally, the `Node` class and the recursive methods **do not** demonstrate OOP style. The recursive methods like `insert(Node)` and `lookup(Node, int)` basically look like recursive functions in any language. In particular, they do not operate against a "receiver" in any special way. Instead, the recursive methods operate on the arguments that are passed in which is the classical way to write recursion. My sense is that the OOP style and the recursive style do not be combined nicely for binary trees, so I have left them separate. Merging the two styles would be especially awkward for the "empty" tree (null) case, since you can't send a message to the null pointer. It's possible to get around that by having a special object to represent the null tree, but that seems like a distraction to me. I prefer to keep the recursive methods simple, and use different examples to teach OOP.

Java Solutions

Here are the Java solutions to the 14 binary tree problems. Most of the solutions use two methods: a one-line OOP

method that starts the computation, and a recursive method that does the real operation. Make an attempt to solve each problem before looking at the solution -- it's the best way to learn.

1. Build123() Solution (Java)

```
/**
 * Build 123 using three pointer variables.
 */
public void build123a() {
    root = new Node(2);
    Node lChild = new Node(1);
    Node rChild = new Node(3);

    root.left = lChild;
    root.right = rChild;
}

/**
 * Build 123 using only one pointer variable.
 */
public void build123b() {
    root = new Node(2);
    root.left = new Node(1);
    root.right = new Node(3);
}

/**
 * Build 123 by calling insert() three times.
 * Note that the '2' must be inserted first.
 */
public void build123c() {
    root = null;
    root = insert(root, 2);
    root = insert(root, 1);
    root = insert(root, 3);
}
```

2. size() Solution (Java)

```
/**
 * Returns the number of nodes in the tree.
 * Uses a recursive helper that recurs
 * down the tree and counts the nodes.
 */
public int size() {
    return(size(root));
}

private int size(Node node) {
    if (node == null) return(0);
    else {
        return(size(node.left) + 1 + size(node.right));
    }
}
```

3. maxDepth() Solution (Java)

```
/**
 * Returns the max root-to-leaf depth of the tree.
 * Uses a recursive helper that recurs down to find
 * the max depth.
 */
public int maxDepth() {
    return(maxDepth(root));
}

private int maxDepth(Node node) {
    if (node==null) {
        return(0);
    }
    else {
        int lDepth = maxDepth(node.left);
        int rDepth = maxDepth(node.right);

        // use the larger + 1
        return(Math.max(lDepth, rDepth) + 1);
    }
}
```

4. minValue() Solution (Java)

```
/**
 * Returns the min value in a non-empty binary search tree.
 * Uses a helper method that iterates to the left to find
 * the min value.
 */
public int minValue() {
    return( minValue(root) );
}

/**
 * Finds the min value in a non-empty binary search tree.
 */
private int minValue(Node node) {
    Node current = node;
    while (current.left != null) {
        current = current.left;
    }

    return(current.data);
}
```

5. printTree() Solution (Java)

```
/**
 * Prints the node values in the "inorder" order.
 * Uses a recursive helper to do the traversal.
```

```

*/
public void printTree() {
    printTree(root);
    System.out.println();
}

private void printTree(Node node) {
    if (node == null) return;

    // left, node itself, right
    printTree(node.left);
    System.out.print(node.data + " ");
    printTree(node.right);
}

```

6. printPostorder() Solution (Java)

```

/**
 * Prints the node values in the "postorder" order.
 * Uses a recursive helper to do the traversal.
 */
public void printPostorder() {
    printPostorder(root);
    System.out.println();
}

public void printPostorder(Node node) {
    if (node == null) return;

    // first recur on both subtrees
    printPostorder(node.left);
    printPostorder(node.right);

    // then deal with the node
    System.out.print(node.data + " ");
}

```

7. hasPathSum() Solution (Java)

```

/**
 * Given a tree and a sum, returns true if there is a path from the root
 * down to a leaf, such that adding up all the values along the path
 * equals the given sum.
 *
 * Strategy: subtract the node value from the sum when recurring down,
 * and check to see if the sum is 0 when you run out of tree.
 */
public boolean hasPathSum(int sum) {
    return hasPathSum(root, sum);
}

boolean hasPathSum(Node node, int sum) {
    // return true if we run out of tree and sum==0
    if (node == null) {

```

```

    return(sum == 0);
}
else {
    // otherwise check both subtrees
    int subSum = sum - node.data;
    return(hasPathSum(node.left, subSum) || hasPathSum(node.right, subSum));
}
}

```

8. printPaths() Solution (Java)

```

/**
 * Given a binary tree, prints out all of its root-to-leaf
 * paths, one per line. Uses a recursive helper to do the work.
 */
public void printPaths() {
    int[] path = new int[1000];
    printPaths(root, path, 0);
}

/**
 * Recursive printPaths helper -- given a node, and an array containing
 * the path from the root node up to but not including this node,
 * prints out all the root-leaf paths.
 */
private void printPaths(Node node, int[] path, int pathLen) {
    if (node==null) return;

    // append this node to the path array
    path[pathLen] = node.data;
    pathLen++;

    // it's a leaf, so print the path that led to here
    if (node.left==null && node.right==null) {
        printArray(path, pathLen);
    }
    else {
        // otherwise try both subtrees
        printPaths(node.left, path, pathLen);
        printPaths(node.right, path, pathLen);
    }
}

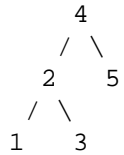
/**
 * Utility that prints ints from an array on one line.
 */
private void printArray(int[] ints, int len) {
    int i;
    for (i=0; i<len; i++) {
        System.out.print(ints[i] + " ");
    }
    System.out.println();
}

```

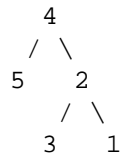
9. mirror() Solution (Java)

```
/**
Changes the tree into its mirror image.
```

So the tree...



is changed to...



Uses a recursive helper that recurs over the tree, swapping the left/right pointers.

```
*/
public void mirror() {
    mirror(root);
}

private void mirror(Node node) {
    if (node != null) {
        // do the sub-trees
        mirror(node.left);
        mirror(node.right);

        // swap the left/right pointers
        Node temp = node.left;
        node.left = node.right;
        node.right = temp;
    }
}
```

10. doubleTree() Solution (Java)

```
/**
Changes the tree by inserting a duplicate node
on each nodes's .left.
```

So the tree...



Is changed to...




```

      / \
     2  3
    /  /
   1  3
  /
 1

```

Uses a recursive helper to recur over the tree and insert the duplicates.

```

*/
public void doubleTree() {
    doubleTree(root);
}

private void doubleTree(Node node) {
    Node oldLeft;

    if (node == null) return;

    // do the subtrees
    doubleTree(node.left);
    doubleTree(node.right);

    // duplicate this node to its left
    oldLeft = node.left;
    node.left = new Node(node.data);
    node.left.left = oldLeft;
}

```

11. sameTree() Solution (Java)

```

/*
Compares the receiver to another tree to
see if they are structurally identical.
*/
public boolean sameTree(BinaryTree other) {
    return( sameTree(root, other.root) );
}

/**
Recursive helper -- recurs down two trees in parallel,
checking to see if they are identical.
*/
boolean sameTree(Node a, Node b) {
    // 1. both empty -> true
    if (a==null && b==null) return(true);

    // 2. both non-empty -> compare them
    else if (a!=null && b!=null) {
        return(
            a.data == b.data &&
            sameTree(a.left, b.left) &&
            sameTree(a.right, b.right)
        );
    }
}

```

```
// 3. one empty, one not -> false
else return(false);
}
```

12. countTrees() Solution (Java)

```
/**
For the key values 1...numKeys, how many structurally unique
binary search trees are possible that store those keys?

Strategy: consider that each value could be the root.
Recursively find the size of the left and right subtrees.
*/
public static int countTrees(int numKeys) {
    if (numKeys <=1) {
        return(1);
    }
    else {
        // there will be one value at the root, with whatever remains
        // on the left and right each forming their own subtrees.
        // Iterate through all the values that could be the root...
        int sum = 0;
        int left, right, root;

        for (root=1; root<=numKeys; root++) {
            left = countTrees(root-1);
            right = countTrees(numKeys - root);

            // number of possible trees with this root == left*right
            sum += left*right;
        }

        return(sum);
    }
}
```

13. isBST1() Solution (Java)

```
/**
Tests if a tree meets the conditions to be a
binary search tree (BST).
*/
public boolean isBST() {
    return(isBST(root));
}

/**
Recursive helper -- checks if a tree is a BST
using minValue() and maxValue() (not efficient).
*/
private boolean isBST(Node node) {
    if (node==null) return(true);

    // do the subtrees contain values that do not
```

```
// agree with the node?
if (node.left!=null && maxValue(node.left) > node.data) return(false);
if (node.right!=null && minValue(node.right) <= node.data) return(false);

// check that the subtrees themselves are ok
return( isBST(node.left) && isBST(node.right) );
}
```

14. isBST2() Solution (Java)

```
/**
 * Tests if a tree meets the conditions to be a
 * binary search tree (BST). Uses the efficient
 * recursive helper.
 */
public boolean isBST2() {
    return( isBST2(root, Integer.MIN_VALUE, Integer.MAX_VALUE) );
}

/**
 * Efficient BST helper -- Given a node, and min and max values,
 * recurs down the tree to verify that it is a BST, and that all
 * its nodes are within the min..max range. Works in O(n) time --
 * visits each node only once.
 */
private boolean isBST2(Node node, int min, int max) {
    if (node==null) {
        return(true);
    }
    else {
        // left should be in range min...node.data
        boolean leftOk = isBST2(node.left, min, node.data);

        // if the left is not ok, bail out
        if (!leftOk) return(false);

        // right should be in range node.data+1..max
        boolean rightOk = isBST2(node.right, node.data+1, max);

        return(rightOk);
    }
}
```