PROGRAMMING WITH C++

(DMCA102)

(MCA)

ACHARYA NAGARJUNA UNIVERSITY

CENTRE FOR DISTANCE EDUCATION

NAGARJUNA NAGAR,

GUNTUR

ANDHRA PRADESH

Lesson 1: Introduction To Object-Oriented Programming

Objectives

After reading this chapter, you will understand:

- The procedure oriented programming concepts and object oriented paradigm.
- The basic concepts of object oriented programming.
- Benefits of object oriented programming.
- Applications of object oriented programming.
- Structure of a C++ program, writing a sample C++ program, compiling and running a program.
- The different types of errors in programming.

Structure Of the Lesson

1.1 Introduction

C++ is an object oriented programming language. It was initially named as C with classes. However in 1983, it was renamed as C++. C++ was developed by Bjarne Stroustrup at AT & T Bell laboratories, New Jersey, U.S.A. It is an enhancement of the C programming language with the major addition of class construct feature of Simula67. It was built upon C and hence all standard C features are also available in C++. Thus C++ is the superset of C. Almost all C programs are also C++ programs.

The three important facilities that C++ have are C with classes, function overloading and operator overloading. These features help to create abstract datatype, inheritance from existing datatype and polymorphism. C++ allows the programmer to build programs with clarity, extensibility and ease of maintenance.

1.1.1 Procedure Oriented Programming

Procedure oriented programming is viewed as a sequence of things to be done, such as reading, calculating, printing etc. A number of functions are written to accomplish these tasks. The primary importance is given to functions and little is given to data that are being used by various functions.

Data is placed as global, so that they may be accessed by all the functions. Each function can have its own data. Global data can be used by any function so that it is difficult to identify what data is used and by which function.

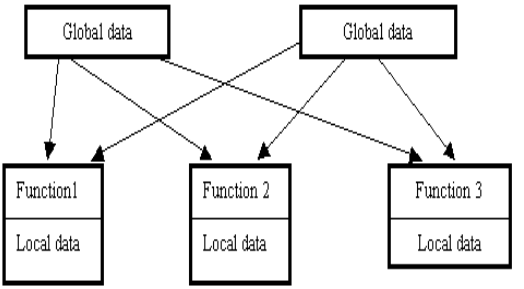
Procedure Oriented programming does not model real world problems very well.

Some Important Characteristics Are:

- Importance is given in doing the algorithms.
- Large programs are divided into smaller programs known as functions.
- Most of the functions share global data.
- Data move over the system from function to function freely.
- Functions transforms the data from one form to another.
- Employs top-down approach in program design.

1.1.2 Object Oriented Programming

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

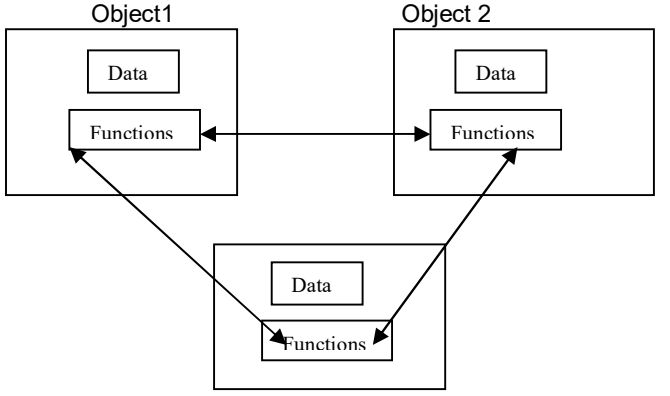


OOPS treats data as a critical element in the program development and does not allow it to freely flow around the system.

It ties data more closely to the function that operates on it and protects it from accidental modifications from outside functions. OOP allows us to decompose the problem into number of entities called objects and build data and function around them. Data of an object can be accessed only by functions associated with that object. Functions of one object can access the functions of another object.

Important Features:

- Importance is given to data then the functions.
- Programs are divided into objects.
- Data Structures characterize the objects.
- Functions that operate on the data of the object are tied together in the data structure.
- Data is hidden and cannot be accessed by the external functions.
- Objects may communicate with each other through functions.
- New data and functions can be easily added.
- Follows bottom-up approach is used in the program design.



Object3

1.2 Concepts Of OOPS

The major Concepts of Object Oriented Programming are:

1. Class
2. Object
3. Abstraction
4. Encapsulation
5. Data Hiding
6. Inheritance
7. Reusability
8. Polymorphism
9. Virtual Functions
10. Message passing

Class: Class is an abstract data type (user defined data type) that contains member variables and member functions that operate on data. It starts with the keyword class. A class denotes a group of similar objects.

e.g.:
```
class employee
{
  int empno;
  char name[25],desg[25];
  float sal;
public:
  void getdata ();
  void putdata ();
};
```

Object: An object is an instance of a class. It is a variable that represents data as well as functions required for operating on the data. They interact with private data and functions through public functions.

e.g.: employee e1, e2;

In the above example employee is the class name and e1 and e2 are objects of that class.

Abstraction: Abstraction refers to the process of concentrating on the most essential features and ignoring the details. There are two types of abstraction

i) Procedural Abstraction
ii) Data Abstraction

Procedural Abstraction: Procedural abstraction refers to the process of using user-defined functions or library functions to perform a certain task, without knowing the inner details. The function should be treated as a black box. The details of the body of the function are hidden from the user.

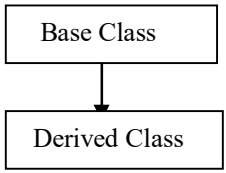
Data Abstraction: Data Abstraction refers to the process of formation of user defined data type from different predefined data types.
e.g: structure, class.

Encapsulation: Encapsulation is the process of combining data members and member functions into a single unit as a class in order to hide the internal operations of the class and to support abstraction.

Data Hiding: All the data in a class can be restricted from using it by giving some access levels (visibility modes). The three access levels are private, public, protected.

Private data and functions are available to the public functions only. They cannot be accessed by the other part of the program. This process of hiding private data and functions from the other part of the program is called as data hiding.

Inheritance: Inheritance is the process of acquiring (getting) the properties of some other class. The class whose properties are being inherited is called as base class and the class which is getting the properties is called as derived class.



Reusability : Using the already existing code is called as reusability. This is mostly used in inheritance. The already existing code is inherited to the new class. It saves a lot of time and effort. It also reduces the size of the program.

Polymorphism: Polymorphism means the ability to take many forms. Polymorphism allows to take different implementations for same name.

poly → many
morphism → forms

There are two types of polymorphism, Compile time polymorphism and run time polymorphism. In Compile time polymorphism binding is done at compile time and in runtime polymorphism binding is done at runtime.

e.g.: Function overloading, operator overloading

Function Overloading: Function overloading is a part of polymorphism. Same function name having different implementations with different number and type of arguments.

Operator Overloading: Operator overloading is a part of polymorphism. Same operator can have different implementations with different data types.

Virtual Functions: Virtual functions are special type of functions which are defined in the base class and are redefined in the derived class. When virtual function is called with a base pointer and derived object then the derived class function will be called. A function can be defined as virtual by placing the keyword virtual for the member function.

Message Passing: An object-oriented program contains a set of objects that communicate with one another. The process of object oriented programming contains the basic steps:

1. Creating classes
2. Creating objects
3. Communication among objects

This communication is done with the help of functions (i.e., passing objects to functions)

1.2.1 Benefits Of OOPS

- Through inheritance, we can eliminate redundant code and extend the use of existing classes.

- Programs can be built from the standard working modules that communicate with one another , rather than writing the code from scratch. This leads to saving of development time and higher productivity.

- Principle of data hiding helps the programmer to build secured programs. It is possible to have multiple instances of objects to co-exist without any inheritance.

- Easy to partition the work in project based objects.

- It is possible to map objects in the problem domain to those objects in the program.

- Software complexity can be easily managed.

- Message passing techniques for communication makes the interface descriptions with external systems much simpler.

- The data-centred design approach enables us to capture more details of a model in implementable form.

1.2.2 Applications of OOPS

The promising areas for application of OOP includes:

- Real-time systems
- Simulation and modeling
- Object-oriented databases
- Hypertext,hypermedia and experttext
- AI and expertsystems
- Neural networks and parallel programming
- Decision support and Automation system
- CIM/CAM/CAD systems

1.3 The Software Life Cycle

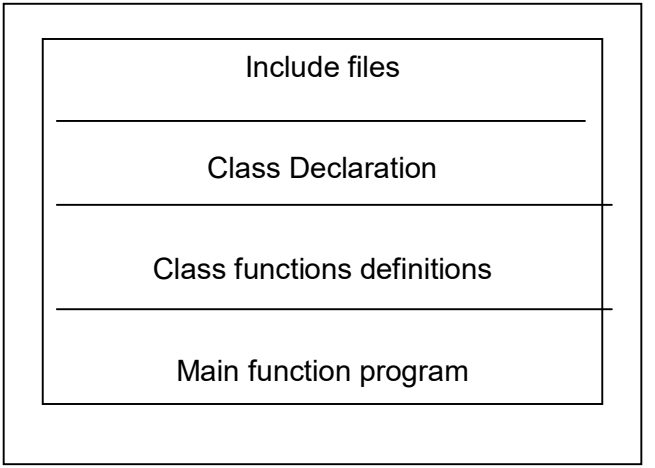
The software development process is divided into six phases known as software life cycle. The six phases of this life cycle are:

- Analysis and specification of the task (problem definition)
- Design of the software (algorithm design)
- Implementation (coding)
- Testing
- Maintenance and evolution of the system
- Obsolescence

1.4 Structure of C++ Program

C++ program contains 4 sections. These may be placed in separate code files and then compiled independently or jointly. A program is commonly organized into 3 separate files. The class declarations are placed in a header file and the definitions of member functions go into another file.

This helps the programmer to separate the abstract specification of the interface (class definition) from the implementation details (member function definition). The main program is placed in a third file which includes the previous two files as well as any other files required.



Structure of C++ program

1.4.1 A Sample C++ Program

```
//sum of two integers
#include <iostream.h>     //include header file
int  main()
{
        int x,y,sum;
        cout <<"Enter any 2 numbers: ";
        cin>>x>>y;
        sum = x + y;
        cout << "The given numbers are ";
        cout<<x;
        cout<<" and ";
        cout<<y;
        cout<<"\n";
        cout<<"Their sum is "<<sum<<"\n";
        return 0;
}               //End of example
```

Output:

```
Enter any 2 numbers: 4 5
The given numbers are        4 and 5
Their sum is 9
```

In order to make a program understandable, some explanatory notes is included at key places in the program. Such notes are called comments. In C++ the symbols // are used to indicate the start of the comments. The comment starts with a // and terminate at the end of the line. A comment may start any where in the line, and whatever follows till the end of the line is ignored. // is a single line comment.

Example:

```
//This is a
//sample C++
//program
```

The C comment symbols /*-------*/ is also valid and are suitable for multiline comments. Either or both of the styles can be used in the programs.

Example:

```
/*This is a       sample C++ program*/
```

The program begins with the line:

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

This is called include directive. It tells the compiler where to find information about certain items that are used in your program. iostream is the name of the library that contains the definitions of the routines that handle input from the keyboard and output to the screen. iostream.h is the file that contains the information about the library.

Directives begin with the symbol # at the very start of the line and no space is included between # and include. A C++ program is a collection of functions. The above example contains one function, main(). The program starts with

```
int main()
{
and ends with
return 0;
}
```

As the return type of the function is integer, 0 is returned here.The lines between the beginning and ending {} are the heart of the program.

```
int x,y,sum;
```

This line is called variable declaration. The variable declaration tells the computer that x,y and sum are the name of the three variables used in the program. int word tells the computer the numbers named by these variables are integers.

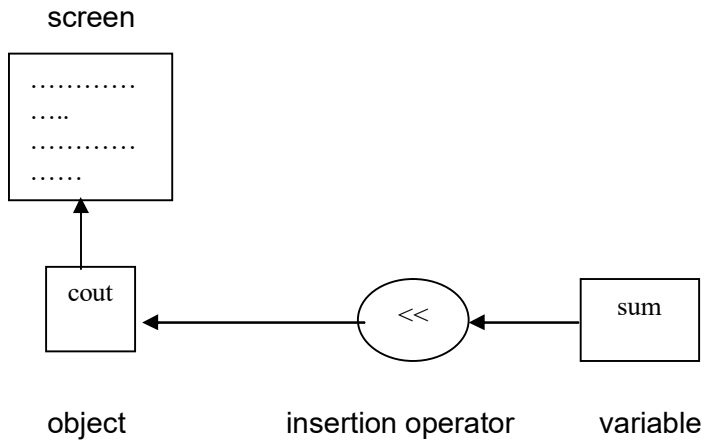
The remaining lines are the instructions that tell the computer to do the corresponding work. These instructions are called executable statements or statements. Every statement should end with a semicolon.

Most of the statements begin with the word cout or cin. These statements are the input and output statements. The << and >> arrows are the operators which tell the direction in which the data is moving.

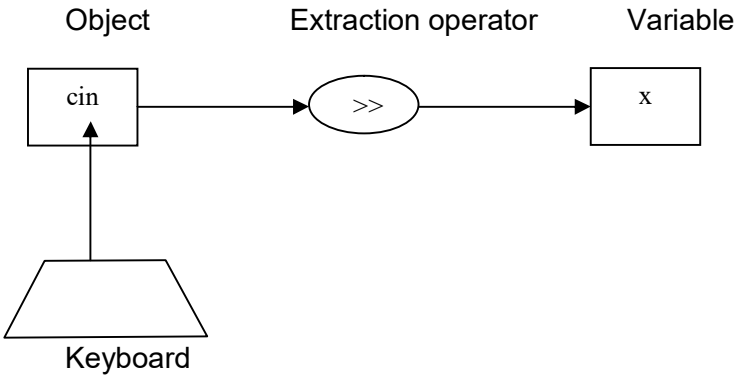
The operator << is called the insertion operator or put to operator. It inserts (or sends) the contents of the variable on its right to the objects on its left. Cout is a predefined object that represents the standard output stream in C++.

Here the standard o/p stream represents the screen. << operator can be overloaded.

Ex: cout<<"Enter two numbers";
cout<<sum;



The operator >> is known as extraction or get from operator. It extracts or takes the value from the keyboard and assigns it to the variable on its right. >> operator can also be overloaded.
e.g.: cin >> x ;



Cascading of I/O operators: The multiple use of << in one statement is called cascading. This is known as cascading of output operator.

e.g.: cout<<"Their sum is"<<sum<<"\n";

This statement sends the string "Their sum is" to cout and then sends the value of sum, then the newline.

Similarly, >> operator can be cascaded. This is known as cascading of input operator.

```
e.g.:   cin>>x>>y;
        sum = x+y;
        cout<<
```

This is the computational statement. The values of x and y are summed up with + operator and the value is stored in the variable sum.

```
        cout<<"Their sum is "<<sum<<"\n";
```

"\n" contained at the end of the output statement tells the computer to start a newline after writing the text.

1.4.2 Compiling and Running

C++ program is typed in using a text editor. There are different text editors. Turbo C++ provides a built-in editor and a menu bar including the options such as File, Edit, Compile and Run. The source file is created and saved under the File Option and can be edited under Edit option. The program is compiled under Compile Option and Run using Run option.

Compilation of the program will produce a machine-language translation of the source code, called the object code. The object code must be linked (combined) with the object code for routines (input and output routines) that are already written. Run option executes the program. If there are no errors in the program, then compiling, linking and running will go smoothly. However, errors may occur which has to be rectified and executed.

1.4.3 Testing and Debugging

A mistake in the program is usually called a bug, and the process of eliminating bugs is called debugging. There are three kinds of programming errors. They are

- Syntax errors
- Runtime errors and
- Logical errors

The errors that are caused due to the violation of syntax (grammar rules) of the programming language are called syntax errors. E.g.: Omitting semicolon at the end of the statement. These errors can be found during compilation.

There are certain kinds of errors that the computer system can detect only when the program is run. These are run-time errors. Eg:If a computer attempts to divide a number by zero.

There are certain kinds of errors, which cannot be identified during compilation. The program is run successfully but the output is wrong. This is due to a mistake in the logic of the program. These are known as logical errors.

E.g.: By mistake, using + instead of * during addition of two numbers.

1.4.4 Applications Of C++

- C++ is a versatile language for handling very large programs.
- C++ is suitable for virtually any programming task including development of editors, compilers, databases, communication systems and any complex real life application systems.
- Since C++ allows us to create hierarchy-related objects. We can build special object oriented libraries which can be used later by many programmers.
- C++ programs are easily maintainable and understandable.

1.5 Summary

Procedure oriented programming follows a Top Down approach where the problem is viewed as sequence of tasks. Functions are used to implement it.

To overcome the drawbacks, such as free movement of data around the program and as it is difficult to model real world problems, object oriented programming is introduced.

Object oriented programming follows a Bottom Up programming approach and it does not allow data to move freely.

The different concepts of OOPS like class, object, encapsulation, abstraction, inheritance, polymorphism, dynamic binding, message passing are briefly discussed.

Advantages and applications of OOPS are discussed.

The structure of a C++ program , writing a sample program, compiling, debugging and running of the programs are discussed.

The applications of C++ are covered.

1.6 Technical Terms

Object : An entity that can store data and, send and receive messages. An instance of class

Class: A group of objects that share common properties and relationships. A class is a new data type that contains member variables and member functions that operate on the variables.

Data Abstraction: The insulation of data from direct access by the programs.

Encapsulation: The mechanism by which the data and functions (manipulating this data) are bound together within an object definition.

Inheritance: Mechanism of deriving a new class from an old class.

Polymorphism: A property by which objects belonging to different classes are able to respond to the same message, but in different forms.

Dynamic Binding: The addresses of the functions are determined at runtime rather than compile time. This is also known as late binding.

1.7 Model Questions

1. Define Procedure oriented programming?
2. Define object oriented programming?
3. What is the difference between object oriented programming and procedure oriented programming?
4. Write the concepts of object oriented programming?

1.8 References

Object-oriented programming with C++
by E. Bala Gurusamy.

Problem solving with C++
by Walter Savitch

Mastering C++
by K.R.Venugopal,
Rajkumar Buyya, T.Ravi Shankar

AUTHOR:

M. NIRUPAMA BHAT, MCA., M.Phil.,
Lecturer,
Dept. Of Computer Science,
JKC College,
Guntur.

Lesson 2: C++ Basics

Objectives

After reading this chapter you will understand:

- About basic terms in C++
- The different data types in C++
- The different types of operators and expressions
- The control structures in C++, Conditional branching, looping and unconditional branching

Structure of the lesson

2.1 Basic terms in C++

Token : The smallest individual units in a program are known as tokens. C++ tokens are

- Keywords
- Identifiers
- Constants
- Strings
- Operators

Keywords have a special meaning in the language. They are reserved identifiers and cannot be used as names for program variables or user-defined variables.
e.g.: int, float, throw, switch ,.. etc.

Identifiers refer to the names of variables, functions, arrays, classes, etc., created by the programmer.

Rules for naming the identifiers :
- Only alphabetic characters, digits and underscores are permitted.
- Name cannot start with a digit.
- Uppercase and lowercase are different(case-sensitive).
- Keyword cannot be declared as variable name.

Constants refer to fixed values that do not change during the execution. They include integers, character, floating point constants and strings.
e.g.:

```
123          //decimal integer
12.34        //floating integer
37           //octal integer
ox2          //hexadecimal integer
"C++"        //string constant
'B'          //character constant
```

2.2 Data types

The data types can be categorized into 3. They are

- Basic or Fundamental data types
- Derived data types and
- User_defined data types.

2.2.1 Basic Data Types

Basic data types are again divided into numeric and non- numeric data types. There are six numeric data types and two non- numeric data types. Out of the six numeric types, three are integer types and three are floating types.

The various integer datatypes are short, int and long . The various floating point data types are float, double and long double. The sizes and the ranges of these data types are follows:

Type name	Memory used	Range	Precision
short (short int)	2	-32768 to 32767	not applicable
int	4	-2,147,483,648 to – 2,147,483,647	not applicable
long(long int)	4	-2,147,483,648 to – 2,147,483,647	not applicable
float	4	10E-38 to 10E18 (approximately)	7 digits
double	8	10E-308 to 10E308 (approximately)	15 digits
long double	10	10E-4932 to 10E4932 (approximately)	19 digits

Note: Precision refers to the number of meaningful digits, including digits in front of the decimal point.

C++ supports 2 non-numeric data types. They are **char** and **bool**. A variable of type char can hold any single character from the keyboard. The value that is stored in a variable of type char are placed within single quotes. One byte of memory is needed for a char type data to be stored in the memory. The data type can be of signed or unsigned char.

e.g: char c1 = 'A';

A new data type called **bool** has been included in C++. It returns a boolean value, true or false with default values 1 and 0.

```
e.g: bool x,y;
     x = true;
```

2.2.2 User Defined Data Types

The user defined data types are structures, unions, class and enumeration.

Structures And Unions: Structures and unions are same as in C. Structures provide a method for packing together data of different types which are logically related.

Eg:

```
struct student
{ char name[20];
int rno;
float tmarks;
};
struct student s1;
```

The keyword **struct** declares student as a new data type that can hold three fields of different data types. These fields are known as **structure members** or **structure elements**. The structure name, student can be used to create variables of type student. s1 is a variable of type student that has 3 member variables.The member variables can be accessed using the dot or period operators.

```
strcpy(s1.name,"John");
s1.rno = 999;
```

Class: A class is a way to bind the data and its associated functions together. It allows data and functions to be hidden if necessary, from external use. Generally , the class specification has two parts.
They are

- **Class Declarations:** It describes the type and scope of its members.
- **Class Function Definitions:** It describes how the class functions are implemented.

General Form of the Class Declaration is:

```
class classname
{
        private:
          variable declarations;
          function declarations;
        public:
          variable declarations;
          function declarations;
};
```

The keyword **class** specifies that what follows is an abstract data type of type classname .The body of the class is enclosed within braces and terminated by a semicolon. The class body contains the declarations of variables or data members and functions or member functions. These functions and variables together called class members. They are grouped under sections as **private** and **public**, which denotes the visibility of the members. The private members can be accessed only within the class and the public members can be accessed from outside the class.
e.g.:

```
class item
        {
        int numb;
        float cost;
        public:
            void getdata(int a,float b);
            void  putdata(void);
        };
```

Once the class is declared, we create objects(variables) of that type by using the classname.

e.g.: item x;
Here, x is an object of the class of item.

Enumerated Data Type: It is a user-defined datatype which provides a way for attaching names to numbers. The keyword **enum** automatically enumerates a list of words by assigning them values 0,1,2, and so on.

e.g.: enum shape{ circle,rectangle,triangle};
Here, circle is assigned an int value 0,rectangle to 1 and triangle to 2.

2.2.3 Derived Data Types

The derived data types are arrays , functions and pointers. We will be discussing about arrays and functions in the next lessons. Pointers are the variables, which directly refer to the value stored in the address it points to.

```
e.g.: int *ip;      //pointer to an integer
      ip = &y;      //address of x assigned to ip
     *ip = 10;
```

2.3 Operators And Expressions

C++ has a rich set of operators. All C operators are valid in C++ also . In addition, C++ introduces some new operators . An expression is a combination of operators, constants and variables arranged as per the rules of the language. The different operators and expressions are studied in this section.

2.3.1 Operators

The operators are used to manipulate the data during the processing. The different types of operators are:

- Arithmetic operators
  There are five arithmetic operators in C++. They are

Operator	Purpose
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	modulus

Syntax:

operand1 operator operand2

Every arithmetic operation returns a numeric value depending on the type of the operands. For the % operator and / operator, the second operator must be non-zero.

Unary Operators: C++ includes a class of operators that act upon a single operand to produce a new value. Other than unary + and unary -, there are two other operators ++ increment,-- decrement operator . These operators can be prefixed(written before) or post fixed(written after) to the operand.

The increment operator causes its operand to be increased by one and the decrement operator causes its operand to be decreased by one. If the operator precedes the operand, then the operand will be altered before it is utilizes called as pre-increment or pre-decrement. If the operator follows the operand, then the value of the operand will be altered after it is utilized called post increment or post decrement.

Bitwise Operators : There are some bit wise operators for the manipulation of data at the bit level. These operators are used for testing or shifting the bits left or right. They may not be applied for float or double values. The bit wise operators and their respective meaning are as follows:

Operator	Meaning
&	bit wise AND
|	bit wise OR
^	bit wise XOR
<<	shift left
>>	shift right
~	one's complement

Relational Operators: C++ supports various relational operators to compare one or more identifiers. The relational operators are:

Operator	Meaning
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	equal to
!=	not equal to

The result of these expressions using relational operators will be boolean values, true represented by 1 or false represented by 0.

Logical Operators: The logical operators are applied between operand or relational expressions resulting in Boolean values true(1) or false(0). The logical operators in C++ are

Operator	Meaning
&&	logical AND
||	logical OR
!	logical NOT

The logical expression yields a value one or zero, depending on the values of the operators and operands according to the truth table given below:

Truth table for And and OR operators:

Op1	Op1	Op1 && Op2	Op1 || Op2
Non zero	Non zero	1	1
Non zero	0	0	1
0	Non zero	0	1
0	0	0	0

Truth table for NOT operator:

Op	!Op
0	1
1	0

Conditional Operators: An operation that makes use of conditional operators(?:) is known as conditional expression. These operations are also known as tertiary operators.

Syntax:

expr1?expr2:expr3

where expr1 is generally a conditional expression and is evaluated first. If it is non-zero then expr2 is evaluated else expr 3 is evaluated.

e.g.:

```
big =(a>b)?a:b;
```

Some other operators that are included in C++ are:

operators	symbols
stream output operator	<<
stream input operator	>>
Scope resolution operator	::
dynamic memory delete operator	delete
dynamic memory allocation operator	new
pointer-to-member operators	::* , ->* and .*

We have already studied about stream input and output operators in the previous chapter. We will study the new and delete operators in the next lessons. Now, we will study about the remaining operators.

Scope Resolution Operator: In C++, blocks and scope are used to construct the program. The scope of the variable extends from the point of declaration till the end of the block i.e., between {}. The same variable names can be used in different blocks to have different meanings. A variable declared inside the block is said to be local to that block. The global version of a variable cannot be accessed from within the inner block. So, a scope resolution operator :: is used , where it allows the access to the global version of a variable.

Syntax:

```
:: variable name
```

Program to demonstrate scope resolution operator

```
#include<iostream.h>
int x = 50;
void main()
{
int x = 10;
  {
        int x = 1;
        cout<<"x = "<<x<<"\n"; //prints x value  local to the scope
        cout<<"::x= "<<x<<"\n"; //prints global x value
  }    cout<<"x = "<<x<<"\n"; //prints x value  local to the scope
    cout<<"::x= "<<x<<"\n"; //prints global x value
}
```

Output:

```
x=1
x=50
x=10
x=50
```

Referencing And De-Referencing Operators: The address operator **&** or referencing operator assigns the address of the operand on the right side to the pointer variable to its left.The indirection operator ***** or de-referencing operator accesses the value of the operand pointed to by the pointer variable.

```
e.g.:   int x = 10,y;
        int *p;
p = &x; //p stores the address of the location where x is stored
y= *p; //*p has the value of the variable to which p points to
cout<<"The value pointed by p is " <<y;
```

Output: The value pointed by p is 10

sizeof Operator: The sizeof operator gives the amount of storage required to store an identifier.

```
int k;
cout<<sizeof(k);
```

Output: 2

Typecast Operator:C++ permits explicit type conversion of variables or expressions using the type cast operator.

Syntax:

```
Typename(expression)
Average = sum/float(i);
```

2.3.2 Precedence Of Operators

Operators in the same box have the same precedence. Operators in higher boxes have higher precedence. Unary operators and assignment operators are executed from right to left and have the same precedence.

Other operators that have the same precedence are executed from left to right. The list is given from higher precedence to the lower precedence.

Precedence of operators are as follows:

::	scope resolution operator
.	dot operator
➔	member selection
[]	array indexing
()	function call
++	postfix increment operator
--	postfix decrement operator
++	prefix increment operator
--	prefix decrement operator
!	not
-	unary minus
+	unary plus
*	dereference
&	address of
new	
delete	
delete[]	
sizeof	
*	multiply
/	divide
%;	remainder(modulo)
+	addition
-;	subtraction
<<	insertion operator(output)
>>;	extraction operator
<	less than
<=	less than or equal
>	greater than
>=;	greater than or equal
==	equal
!=;	not equal
&&;	and

|| ;	or

=	assignment
%=	modulo and assign
+=	add and assign
-=	subtract and assign
*=	multiply and assign
/=;	divide and assign

2.3.3 Expressions

Expressions combine operands, operators and constants to produce a single value. There are different types of expressions like:

Constant Expression: They can have only constants values.

e.g.:10
15+6/4.0
y'

Integral Expressions: These expressions produce integer result after implementing all automatic and explicit type conversions.

e.g.:X
X * 'a'
5 + int(2.0)
where X is an integer

Float Expressions: These expressions produce floating point results after implementing all automatic and explicit type conversions.

Eg:
X
X * y/10
5 + float(2)
where x and y are floating point values.

Pointer Expressions: These produce address values.

e.g.: &x
ptr
ptr + 1
where x is a variable and ptr is a pointer

Relational Expression: These expressions results in a bool type values, true or false.

e.g.: x<y
a+b>100

Logical Expressions: These combine two or more relational expressions using logical operators and result a bool type values.

e.g.: i<j && y == 5
p == 3 || q >5

Bitwise Expressions: Bitwise expressions are used to manipulate data at bit level.

e.g.:x<<3;//shifts 3 bits to its left

Special Assignment Expressions: Chained assignment:

A = (b = 5);
Or
A = b= 5;

First 5 is assigned to b, then to a.
A chained statement cannot be used to initialize variables at the time of declaration.

Embedded Assignment:

x = (y = 20) + 10;

(y = 20) is an assignment expression known as embedded assignment. The value of 20 is assigned to y and then the result 20+10 is assigned to x.

Compound Assignment: Compound assignment operator is a combination of assignment operator with a binary arithmetic operator.

p = p + 10;
can be written as
p+=10;

2.4 Control Structures

In high-level programming languages, flow of program execution may be changed using certain control statements called control structures.

A control structure is a **control flow statement** that allows you to alter the **sequential flow**.

Control flow statements fall into three categories:

1. Conditional branching (or) Decision Making or Non-iterative
2. Looping or iterative or repetitive
3. Unconditional branching.

2.4.1 Conditional Branching

Conditional branching is the most basic control feature of any programming language. It enables a program to make decisions, to decide whether or not to execute a statement or a block of statements based on the value of an expression. The expression may result in either true or false value. Since the value of the expression may vary from one execution to another, this feature allows a program to react dynamically to different data.

C supports various types of conditional branching statements. The following categories illustrate several conditional control structures.

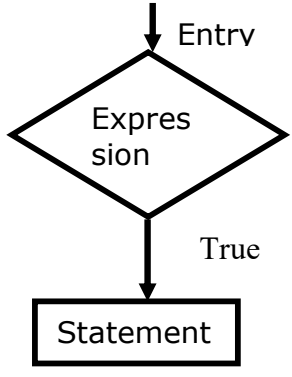
- **Simple if**
- **if ..else**
- **else if ladder**
- **Nested if**
- **Switch**

Simple If: The **simple if** statement is wonderful decision making statement and is used to control the flow of execution of a single or multiple instructions.

The general form of **"simple if"** follows:

If (condition/expression) Statement;

In this statement the given condition is tested first and responds accordingly. If the result of expression is true then the given statement is executed. If the result is false the statement cannot be executed.



When multiple statements are to be executed using if control structure then it may be referred as compound if.

Syntax:

```
if (expression)
{
        statement-block;
}
statement-x;
```

The statement-block may be a single statement or a group of statements. If the expression is true statement-block will be executed, other wise the statement-block will be skipped and the execution will jump to the statement-x.

Program to find biggest of two numbers.

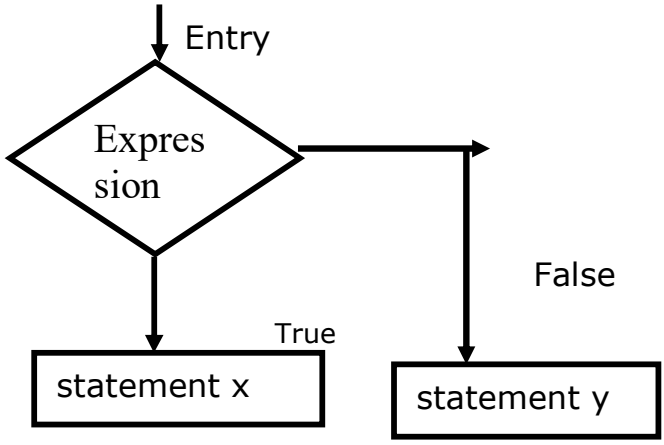
```
#include<iostream.h>
void main()
{
        int a, b;
        cout<<"\n\t Enter A value              : ";
        cin>>a;
        cout<<"\n\t Enter B value              : ";
        cin>>b;
        if (a>b)
                cout<<"\n"<<a<<" is Greater than "<<b;
        if (b>a)
                cout<<"\n"<<b<<" is Greater than " <<a;
}
```

If_else: In **if-else** control statement there exists an extension of the simple if statement. It allows the user to perform another block of statements in case the condition result is false.

syntax :

if (expression)
statement-x;
else
statement-y ;

Here the **expression** is evaluated; if the result of the expression is a true then statement-x is executed otherwise statement-y will be executed.



Flow graph

Program to check whether given number is even or odd

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

        cout<<"\n Enter a number..:";
        cin>>n;
        if (n%2==0)
                cout<<"\n Given  number is even":
        else
                cout<<"\n Given  number is odd":

}
```

else-if Ladder: In **else..if ladder** number of conditions are checked depending on the falsity of the previous condition. Literally, too many conditions are evaluated in **if**..**else** ladder.

Syntax:

```
If <condition1>
{
------
}
else if <condition2>
{
------ True block 1
}
else
{
------ False block
}
```

In this, condition1 is checked and if it is true then its corresponding condition is executed. If the condition is false then next condition is verified. If all the given conditions are false then false block is executed. Only one of all the available blocks gets executed. After the execution of any one of the blocks, control is transferred to next statement after the construct.

Program to find biggest of three numbers

```
#include<iostream.h>
void main()
{
        int a,b,c;
        clrscr();
        cout<<"enter three numbers:";
        cin>>a>>b>>c;
        if(a>b)
                if(a>c)
                        cout<<a<<" is big";
                else
                        cout<<C<<"is big";
        else if(b>c)
                cout<<b<<" is big";
        else
                cout<<c<< " is big ";
}
```

Decision Making With Nested If: A **nested if** control structure consists of multiple **if** statements in one another. Here each **if** statement consists of subsequent branching statement. Literally a nested **if** consists of one **if** statement in another **if** statement. It is used when multiple conditions are to be evaluated.

Syntax:

```
        if(expression)
{
        if(expression)
        {
                if(expression)
                {

                        ---
                        ---
```

Here evaluations of expressions or conditions are based on the first condition. If the first condition itself is false, then there is no way of evaluating other conditions. At any level of expression the program control may be altered.

Ex: Program Biggest of 3 numbers using nested if

```
#include<iostream.h>
void main()
{
        int a,b,c,big;
        cout<<"\n Enter the value of a  :  ";
        cin>>a;
        cout<<"\n Enter the value of b  :  ";
        cin>>b;
        cout>>"\n Enter the value of c  :  ";
        cin>>c;
        if (a>b)
                if (a>c)
                        big = a;
                else
                        big = c;
        else
                if (b>c)
                        big = b;
                else
                        big = c;
        cout<<"\nBiggest of three numbers is:"<<big;
```

}

switch: C provides a special kind of conditional control structure that acts as an alternative to **if..else ladder**. When there are more conditions or paths in a program, **if-else** branching can become more difficult. In such situations **switch** may act better. The **switch** statement allows the user to specify an unlimited number of execution paths based on the value of a single expression. Each execution path is referred as a case.

However, all the cases should be unique. Each case must be terminated by a '**break**' statement. The '**default'** case is not mandatory.

In a **switch** statement, there are four different keywords to be used:

- **switch**
- **case**
- **break**
- **default**

Though the **switch** control structure enables the user to improve clarity of the program, it causes more errors. So, it requires more attention while implementation.

Syntax:

```
switch(expression)
        {
        case  value1:
                        statement;
                         break;
        case  value2:
                        statement;
                         break;
        :
        :
        :
        default :
                       statement;
        }
```

Among all the cases, only one case can be executed successfully because each case is terminated by a '**break'** statement.

Program to accept two integer values and perform arithmetic operation by getting the user input.

1) Addition 2) Subtraction
3) Multiplication 4) Division
5) Exit .

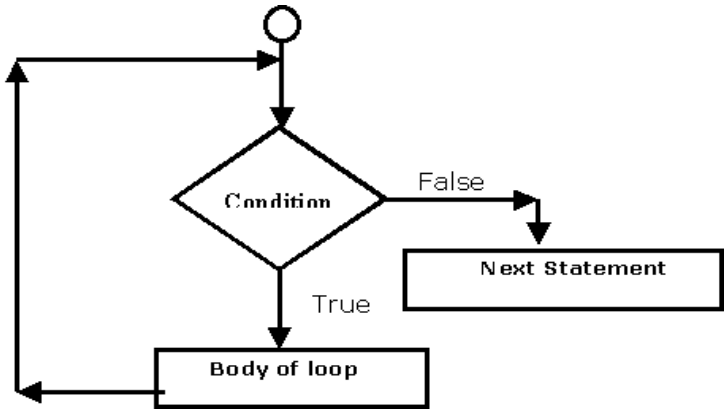
```
#include<iostream.h>
#include<conio.h>
#include<stdlib.h>
void main()
{
        int a, b, c, ch;
        clrscr();
        cout<<"\n\t\t\t Enter two numbers     :       ";
        cin>>a>>b;
        cout<<“Enter your choice:”:I
        cout<<“1)Addition\n2)Subtraction”;
        cout<<“\n3)Multiplication”;
        cout<<”\n4) Division. \n5) Exit”.
        cin>>scanf(“%d”&ch);
        switch (ch)
        {
        case 1:
                c = a + b;
                break;
        case 2:
                c = a - b;
                break;
        case 3:
                c = a * b;
                break;
        case 4:
                c = a / b;
                break;
        default :
                cout<<"\n Invalid option  ";
                exit(0);
        }
        cout<<"\n\t\t\t Result   :”,c;
}
```

2.4.2 Looping Structures

Some times, in a program, a statement or a block of statements need to be executed repeated number of times. In such situations decision control structures may not be useful, as they do not transfer the control back. Hence the user may require another form of control structures, which perform a group of instructions for a fixed number of times. Such control structures are named as looping control structures. C language provides three different iterative or looping structures.

- **while** loop
- **do...while** loop
- **for** loop

While: The **while** control structure executes a single or multiple statements for repeated number of times based on a given condition. It executes the statements as long as the given condition or expression results in a true value. It terminates execution as and when the condition is false.



Syntax:

```
initialization statement;
while(condition)
{
    --------------
   Condition reachable Statement;
}
```

Here the condition is tested every time, it executes the block of statements. The keyword **while** verifies the trueness and falsity of the expression and responds accordingly. If the condition is false for the first time the minimum number of iterations is 0 in **while** control structure. It requires three statements in order to perform repetitive tasks.

They are

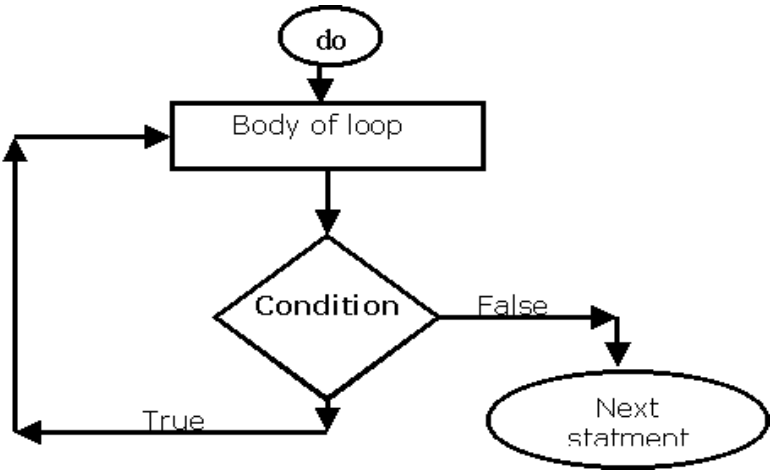
- Initialization statement
- Conditional statement
- Condition reachable statement

If any of the above statements is ignored then the **while** may not perform well.

Program to print the numbers from 1 to 10

```
#include<iostream.h>
void main()
{
  int i;
  i=1;
  while (i<=10)
    {
      cout<<”n"<< i;
      i++;
    }
}
```

Do- While: C provides another form of while control structure i.e., **do-while** control structure. In **do-while** control structure the statements in the block get executes first, later on the condition is evaluated. Hence the user can assume that the minimum number of iterations for **do-while** control structure as 1, even if the expression or condition results in false for the first time.



Syntax:

```
Initialization statement;
do
{
-------------
Condition reachable statement;
} while(condition);
```

Here the statements in the loop will be executed until the given condition becomes false. The while statement should be terminated by a semicolon (;) in **do while**.

Print the numbers from 1 to 10.

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

For Loop: C provides a more flexible form of looping control structure that improves clarity of the code. It is nothing but **for** control structure. Usually the **for** control statement is used to perform fixed number of iterations.

The major difference between **for** and **other looping structures** is the number of iterations. In case of **while** and **do-while** the number of iterations are indefinite. The user may not predict the number of iterations. On the other hand **for** specifies the number of iterations in the statement itself.

Syntax:

for *(initialization; test condition; increment/decrement part)*

```
{
        Body of the loop;
}
```

The initialization may contain single or multiple assignment statements. A control variable is involved in this part of statements. The test condition verifies the validity of the control variable for each iteration. Increment or decrement part, increments or decrements the value of the control variable in order to reach the test condition.

Program to print the numbers from 1 to 10.

```
#include <iostream.h>
void main()
{
  int i;
  for (i=1 ; i<=10; i++)
    cout<< I<<"\n";
}
```

Break and **Continue** statements:

Break: This statement takes control out of the **switch** statement or loop structure. In other words, a **break** statement takes the control out of the current block in execution. The control is transferred to the statement that follows the block.

Syntax:

```
break;
```

Continue Statement : To skip a part of the body of the loop in execution on certain condition and for the loop to be continued for the next iteration **continue** statement is used.

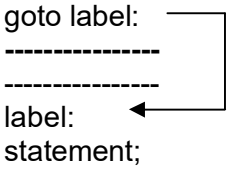
Syntax:

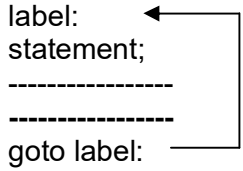
```
continue;
```

2.4.3 Unconditional Branching

goto and label: C++ supports an unconditional branching statement called **goto**. This **goto** is meant for transferring control from one part of the program to another part a label is present. A label is a user-defined word to where the control is supposed to be transferred. The given label must reside in the same function and can appear before only one statement in the same function. Although it may not be preferable to use the **goto** statement in a highly structured language like C, there may be occasions where the use of **goto** is desirable.

Syntax:





example demonstrates the **goto** statement:

```
void main()
  {
     int x = 1;
     abc:
          cout<<x;
          x++;
          if(x <= 5 )
            goto abc;
   }
```

2.5 Summary

- The basic terms in C++ like token, keyword, identifier, constants are studied here.
- The data types in C++: basic, user defined and derived data types are discussed in detail.
- The different types of operators and expressions are also discussed in this lesson.
- The different control structures in C++ like conditional, looping and unconditional statements are studied. The conditional statements: if, if-else, nested If and switch are studied in detail. Also focus is made on three categories of loops, available in C++ language: **while**, **do–while** and **for** loop. Usage of break, continue, **goto** and exit statements, which are very useful in loops have been covered.

2.6 Technical Terms

Expression: It is a combination of operators, constants and variables arranged as per the rules of the language.

Operator: A symbol that represents an action to be performed.

Manipulator: A data object that is used with stream operators. It causes a specific operation to be performed on the stream.

Scope resolution operator: The operator that is usually used to indicate the class in which the identifier is declared.

Type casting: To convert a variable from one type to another type by explicitly.

Union: A data type that allows different data types to be assigned to the same storage location.

2.7 Model Questions

1. Explain the different data types in C++ ?
2. What are the different operators in C++ ?Explain.
3. Explain the precedence of operators.
4. How many types of expressions are there? What are they ? Explain them ?
5. Explain the different control structures in C++ with example ?

2.8 References

Object-oriented programming with C++
by E.Bala Gurusamy

Problem solving with C++
by Walter Savitch

Mastering C++
by K.R.Venugopal, Rajkumar Buyya, T.Ravi Shankar

AUTHOR:

M. NIRUPAMA BHAT, MCA., M.Phil.,
Lecturer
Dept. Of Computer Science
JKC College
GUNTUR.

Lesson3: Procedural Abstraction And Functions

Objectives

After completing this lesson you will understand about:

- The functions and types of functions.
- How to write functions.
- Local variables, void functions, function overloading, inline functions.
- Parameter passing, passing default arguments and function calling another functions.
- Recursive functions
- Procedural abstraction, testing and debugging.

Structure Of The Lesson

3.1 Functions

C++ has facilities to include separate subprograms into programs. These subprograms are called functions. A function that returns a value is like a small program. The arguments to the function serve as the input to this small program and the value returned is like output of this program. When a subtask of a program take some values as input and produce a single value as its result, then such subtask can be said as a function.

Functions can be divided into two. They are

a) Predefined functions
b) Programmer defined functions.

3.1.1 Predefined Functions

Predefined functions are the functions that are already built and supplied along with the compiler. These functions are also called library functions and they are stored in the header files with ".h" extension. A header file for a library provides the compiler with certain basic information about the library and **include** directive delivers this information to the compiler. Some of the header files are – iostream.h, math.h, ctype.h,stdlib.h, string.h, manip.h, conio.h some predefined math functions are:

Return type	Function name	Example	O/p
double	sqrt(double n)	sqrt(4.0)	2.0
This returns the square root of a given number			
double	pow(double a,double b)	pow(2.0,3.0)	8.0
This returns the a^b . If the first argument is a negative number, then the second argument must be a whole number.			
double	fabs(double num)	fabs(-7.5)	7.5
It calculates the absolute value of the given number			

double	ceil(double num)	ciel(3.2)	4

It returns the smallest integer greater than the number.

double	floor(double num)	floor(3.9)	3

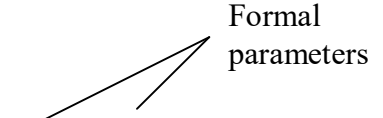
It returns the largest integer less than the number.

3.1.2 Programmer Defined Functions

A function can be defined by the user either in the same file as part of the main program or in a separate file so that the function can be used several times. Such functions are called **Programmer defined functions**. A function is like a small program is same as running the program. A function generally uses formal parameters to input the various values into the function. The description of the function is divided into two parts. They are called

- function prototype
- function definition

Function Prototype: The prototype describes what the function look like i.e., it tells about the function name the return type of the function, the number of arguments and the type of the arguments passed into the function, the identifier name may or may not be declared in the arguments list of the prototype. The identifiers used in the arguments list are called as formal parameters. The formal parameters are used as a kind if place holder for the arguments. The general format of the function prototype is:



Syntax:
Type returned function name(type1 fp1,jtype2 fp2);

e.g.: double totalweight (int num, double weight);
or
double totalweight (int, double);

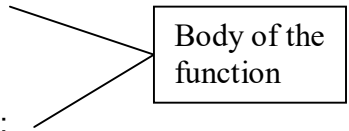
A function prototype should appear before the function call, normally they are placed before the main part of the program.

Function Definition: A function definition describes how the function computes the value it returns. A function definition consists of a function header followed by the function body. A function header is written same as the function prototype, except the header does not have a semicolon at the end. The function body follows the function header and completes the definition. The function body consists of declaration and executable statements enclosed within a pair of braces. the value returned by the function is determined when the function executes a return-statement.

The general format of a function definition is given below.

```
Return type function name (data type fpar1,datatype,fpar2…)
{
declaration
:
:
executable statements

:
:
return statement;
}
```

```
eg:  float billcal(int n, float c)  //function header
{
const float tax=0.15;
float bill amount;
billamount=n*c;
bill=billamount+billamount*Tax;
return bill;
}
```



Function Call: Function call is a statement that involves a controlled transfer to the definition of the function, i.e., invokes the function. The function call statement consists of function name and list of actual arguments written within the parameters.

Syntax: Function name (actual parameters list);

Write a program to display maximum of two numbers using functions.

```
#include<iostream.h>
void main()
{
int a,b;
int max(int,int);
cout<<"enter a,b";
cin >> a>>b;
cout<<maxi(a,b);
getch();
return 0;
}
int max(int a, int b)
{
if(a>b)
return a;
else
return b;
}
```

3.1.2 Local Variable

Variables that are declared within the body of the function are said to be local to that function or the scope is within the function. The variables that are declared in the main part of the program are said to be local to the main part of the program.

If variable is local to a function then we can have another variable with the same name which is declared in the main part of the program or in some other function. These two will be different variables even though they have the same name.

Program to demonstrate the local and global variables

```
#include<iostream.h>
#include<conio.h>
int x=5;//global variable
int fun1( );
int fun2( );
int fun3( );
```

```
void main( )
{
cout<<x<<end1;
int  x=10;        //local to main
cout<<fun1( )<<end1;
cout<<fun2( )<<end1;
cout<<fun3( )<<end1;
cout<<x;
return 0;
}
int fun1()
{
x+=2;   //uses global variable
cout<<x<<"\n";
return x;
}
int fun2( )
{ int x;  //local to fun2()
cin>>x;
cout<<x<<"\n";
return x;
}
int fun3( )
{
 x=x+1;  //uses global variable
cout<<x<<"\n";
return x;
}
```

3.3 Parameter Passing Into Functions

The parameters to a function can be passed in two ways. They are

- Call by value
- Call by reference

Call By Value Mechanism: Formal parameters of a function are local to the function. They can be used just like a local variable. We should not add a variable declaration for the formal parameters. When a function is called the values of actual arguments are copied to formal parameters within the function. We can make any changes to the values of the formal parameters but these changes will not be reflected on the values of the actual arguments. This mechanism is called call by value mechanism.

Program to demonstrate call by value mechanism

```
#include<iostream.h>
void swap (int,int);
void main()
{
int a,b;
cout<<"enter a,b";
cin >> a>>b;
swap (a,b);
}
void swap(int a, int b);
{
int c;;
c=a;
a=b;
b=c;
cout<<a<<" "<<b;
}
#include<iostream.h>
void swap (int,int);
```

output:

```
enter a,b2
3
3 2
```

Call By Reference Mechanism: To make a formal parameter a call by reference parameter an "&" sign should be appended to its date type name. The corresponding actual argument in the function call should be a variable but not a constant or an expression when the function is called the corresponding actual variable argument (not a value) will be substituted for the formal parameter. Any changes made to the formal parameter in the function body will reflect on the actual argument variable.

Program to demonstrate call by reference mechanism

```
#include<stdlib.h>
#include<iostream.h>
#include<iostream.h>
void read(int &x);// prototype type for read function.
void main()
{
int a,b;
cout <<"enter two integers:";
read(a);
```

```
read(b);
cout<<"a is " <<a<<end1;
cout<<"b is"<<endl;
}
void read(int&x)
{
cin >>x;
}
```

output:

```
enter two integers:3
3
a is 3
b is
```

Program to read two variable and arrange them in order using 3 functions read (), swap (), display().

```
#include<iostream.h>
#include<conio.h>
void read(int &x,int &y);
void swap(int &x,int &y);
void show result(int x,int y);
void main()
:
int a,b;
cout <<"enter a&b values :";
read (a,b);
if(a>b)
swap(a,b);
showresult(a,b);
getch();
}
void read (int &x, int &y)
{
cin>>x>>y;
return;
}
void swap(int &x,int &y)
{
x=x+y;
y=x-y;
x=x-y;
return;
}
void show (int x, int y)
```

```
{
cout<<"after swaping:"
<<a<< end <<b;
return;
}
```

output:

```
enter a&b values :3 4
after swaping: 4 3
```

3.4 Function Overloading

Writing two or more function definitions with the same function name is called function overloading. Such overloading definitions must have different number of formal parameters or it can have same number of parameters with different data types. When there is a function call, the compiler uses the function definition whose number of formal parameters and the type of parameters match the argument in the function call.

C++ allow us to use the function name by overloading them .However, it is not possible to overload a function name by giving two functions that differ only in the type of return value. I.e., the overloaded definitions are uniquely identified with the help of arguments list for the functions.

3.5 Void Functions

Sub tasks are implemented as functions in C++. A subtasks might produce several values are it might return no value. A function must either return a single value or return no value at all. A function that return no value is called a void function. In the void function it does not return any value to the rest of the program but produces the output on the screen.

Syntax:

```
//Function prototype
void function name(parameter list);
//function definition.
Void function name(parameter list)
{
body of the function;
```

```
:
:
return;
}
```

Program to demonstrate void functions

```
#include<iostream.h>
void starline(void) // this prints line of 10 *'s
{
for ( int  i=1; i<=10;i++)
cout<<"*";
return;
}
void starline(int n)
{
for (int i=1;i<=n;i++)
cout<<"*";
return;
}

#include<iostream.h>
void main()
{
starline();
int n;
cout<<"enter n value";
cin>>n;
starline(n);
}
```

output:

```
enter n value20
```

Void function may not have any parameters at all. It is optional to write a return statement in your void function. However, a return statement is used to make a conditional exit.

```
void print quotient(int x, int y);
{
if (y==0)
return;
cout <<”quotient=”<<x/y;
}
```

3.6 Function Calling Another Function

A function body can contain a call to another function. When this inner function is called its prototype should appear before its first use. Although the function call is included in the definition of another function, the definition of the called function should be put outside the main program(definition).

e.g:

```
void order (int & n1, int&n2)
{
        if (n1>n2)
        swap (n1,n2);
}
}
void swap (int n1, int &n2)
{
        int temp
        temp = n1;
        n1 = n2;
        n2 = temp;
        return;
}
```

3.7 Inline Functions

When a function is called lot of extra time is taken in jumping to the function saving registers pushing arguments into stack and returning into the calling function for very small function this process increases the overheads (time). To eliminate the cost of calls to small functions a new feature called in line function is proposed. An inline function is a function i.e. expanded in line when it is invoked. The compiler replaces the function call with the corresponding function code (similar to macro expansion)

Syntax:

```
inline function name(parameter list)
{
function body
}
```

3.8 Default Arguments

C++ allows a call to the function without specifying all its arguments. In such conditions, the function assigns a default value to the parameter which does not have a matching argument in the function call.

Default values are specified when the function is declared. The compiler looks at the prototype uses and allots the program for possible default values we must add default from right to left in proto typing default arguments are useful where some arguments always have the same value.

program to demonstrate the usage of default arguments

```
#include<iostream.h>
float value (float p, float t, float r=0.15);
void printline(char ch='*', int len=40);//Default arguments
int main()
{
float amount;
printline();
amount=value(5000,0.5);
cout<<"Amount = "<<amount;
return 0;
}
float value (float p, float t, float r)
{
float amount;
amount=(p*t*r)/100.0;
return amount;
}
void printline(char ch, int len)
{
for ( int i=0; i<=len;i++)
cout<<ch;
cout<<"\n";
}
```

output:

```
**********
Amount = 3.75
```

3.9 Recursive Function

A function that contains a function call to itself, or a function call to a second function which eventually calls the first function is known as a recursive function . The recursive definition for computing the factorial of number can be expressed as follows :

fact (n)= { 1 if n=0

N*fact (n-1), otherwise

Recursion, as the name suggests, revolves around a function recalling itself . The recursive approach of problem solving substitute the given problem with another problem of the same form in such a way that the new problem is simpler than the original .

Two important conditions which must be satisfied by any recursive function are :

1. Each time a function calls itself it must be nearer, in some sense, to a solution.
2. There must be a decision criterion for stopping the process or computation .

Recursive function involves saving the return address, formal parameters, local variable upon entry , and restore these parameters and variable on completion.

factorial of a number using recursion

```
#include<iostream.h>
void main (void)
{
     int n;
     long int fact(int);    //prototype
     cout<<"Enter the number whose factorial is to be found
";
     cin>>n;
      cout<<"The factorial of "<<n<<"is"<<fact(n)<<endl;
}
long fact( int num )
```

```
{
    if(num==0)
       return 1;
     else
        return num *fact (num -1);
}
```

output:

Enter the number whose factorial is to be found 5
The factorial of 5is120

3.10 Procedural Abstraction

The principle of procedural abstraction is that, the function should written so that it can be used like a black box. This means that the programmer who uses the function need not know the body of the function to see how the function works. The function prototype and its comment should be known in order to use the function.

How To Write A Black Box Function Definition : The prototype comment should tell the programmer all the condition that are require as argument the function and it should describe the value returned by the function when the function is called.All variables used in the function body should be declared in the function body itself (formal parameters need not be declared).

The function prototype is broken down into two kinds of information called a pre- condition and post- condition. Pre – condition states that the assumption to be true when the function is called .The function should not be used and can't be expected to perform correctly unless the pre-condition holds. The post-condition describes the effect of the function call. i.e.., what will be true after the function is executed, when the pre-condition holds.

e.g.: void swap(int & var1 , int& var2);
// Pre-condition: var1, var2 has been given values
// Post condition: The value of var1, var2 has been changed

Note: Designing a function that can be used a black box can also be called as information hiding .

3.11 Testing And Debugging

In top-down design each function is designed, coded (C++ program) and tested as a separate unit from the rest of the program. The given task is divided into smaller and manageable subtask. These subtasks are converted into C++ functions. To test each of the functions independently, drivers and stubs are used.

A **driver** program is a temporary tool with minimum code. This program gets values from the function arguments in the simplest possible way. It need not do all the calculations that the final program performs.

This program contains some loop statements to repeat the test with different values. Sometimes it is not possible to test a function without using some other functions output which was not yet tested.

In this case a simplified version of the untested function called a **stub** is used. A stub need not perform correct necessary value for testing in the simplest possible way.

Rules And Methods Of Testing: Every function should be tested in a program. The common approach for testing basic functions like I-O is using the driver programs. We use stubs to test the remaining functions.

The stubs are replaced by functions one at a time, i.e., replace by its function def and tested once the function is fully tested another stub is replaced and this process continues until a final program is produced.

3.12 Summary

- The function is defined and the types of functions , predefined and programmer defined functions are discussed.
- Call by value and call by reference mechanism in parameter passing to functions are studied.
- Details of Function overloading, void functions are made clear.

- Calling a function within a function, passing default arguments to the functions are discussed.
- Procedural abstraction and writing stubs and driver programs while testing and debugging are covered.

3.13 Technical terms

Actual Arguments: Arguments that are used in the function call.

Formal Arguments: Arguments that are used in function header. They are placeholders that is filled with function arguments when the function is called.

Function declaration: This provides the information for the function call. This is also known as function prototyping.

Function definition: It describes how the function computes the value it returns.

Function Overloading: A language feature that allows a function to be given more than one definition. The types of arguments with which the function is called, determines which definition is used.

Information Hiding: The hiding of the state and its implementing details in the module.

Inline functions: A function definition such that each call to the function is replaced by the statements that define the function.

3.14 Model Questions

1. Discuss the different types of functions?
2. What are predefined and programmer defined function? Explain with examples.
4. Explain the different mechanisms in parameter passing.
5. What are void functions? Explain with example.
6. What is an inline function? Explain with example.
7. What is a driver ?
8. What is a stub?
9. Explain the principle of procedural abstraction.

3.15 Reference Books

Object-oriented programming with C++

by E.Bala Gurusamy

Problem solving with C++

by Walter Savitch

Mastering C++

by K.R.Venugopal, RajkumarBuyya, T.RaviShankar

AUTHOR:

M. NIRUPAMA BHAT, MCA., M.Phil.,
Lecturer
Dept. Of Computer Science
JKC College
GUNTUR.

Lesson 4: I/O Streams

Objectives

After completing the lesson you will understand about:

- The streams, advantages of using files and about file input and output.
- The different functions in file I/O.
- The tools for formatting the output functions and manipulators
- The different character functions used in I/O and predefined character functions.
- The different file modes, file pointers and file manipulation.
- Sequential and random access files.

Structure Of The Lesson

4.1 Streams

Stream is a flow of characters or data. If the flow is into the program then it is called input stream. If the flow is out of the program it is called an output stream. cin is an input stream object connected to the keyboard. cout is an output stream object connected to the screen. These two streams are automatically available to the program through the header file <iostream.h>. However, we can define stream that come from or go to the files. These streams are called file streams.

4.1.1 Advantages Of Using Files For I-O

The keyboard input and the screen output deals with temporary data. When the program ends, the data typed in at the keyboard and the data left on the screen go away. If we want to store data permanently then we should store it in a file. The contents of the file remain until it is modified by a person or a program. If the output from the program is sent to a file it will remain there even after the program has finished running. An (input) file can be used any number of times, by any number of programmers.

4.1.2 File I-O

When the program takes input from a file, it is said to be reading from the file. When the program sends output to a file it is said to be writing to a file. In C++, a stream is a special kind of variable known as an object.

The stream objects are used to read from and write into the files. An input stream that reads from a file is called input file stream objects and output stream that writes to a file is called output file stream object. An input file stream object is created using the class "**ifstream**" and output file stream object is created using the class "**ofstream**". The two types are defined in the library with the header file "**fstream.h**". Hence a program using these two types must include "fstream.h".

ex: for stream variables
```
ifstream   instream, fin;
ofstream   outstream, fout;
```

instream, fin are ifstream objects. outstream, fout are ofstream objects.

A stream object should be connected to a file after it is declared. This is called opening of a file and it is done with a function name “open”. The syntax of opening the file is

i/o streamobject.open(“file name”);
e.g.: fin.open(“in file.dat”);

After connecting an input file stream object, we can access data from the input file using the stream object and extraction operator.
fin >> y;

An output stream is also opened in the same way.
e.g: fout.open(“outfile.dat”);

The open function in the above example will create the output file, outfile dat, if it does not exist. If the outputfile already exists, the open function discards(removes) the contents of the file. Every input and output file used in the program has two names, the external name and the internal name. The external filename is the real name of the file that is used as an argument in the “open” function. The stream object connected to this file is the internal name of the file. This internal name is used in rest of the program.Every file should be closed when the program has finished getting input or sending output to the file. Closing a file disconnects the stream from the file. A file is closed with a function called close. The close function does not require any arguments.

Syntax:

```
        i/p streamobject.close( );
        o/p streamobject.close( );
eg:-    fout.close();
        fin.close();
```

Sample program using file concept to store the data and keyboard to input the data

```
#include<iostream.h>
#include<fstream.h>
void main()
{
int x,y,z;
ofstream fout;
fout.open("ex1.dat");
cout<<"enter 3; integers:";
cin>>x>>y>>z;
fout<<"The 3 integers are: ";
fout<<x<<"  "y<<" "<<z;
fout.close();
return;
}
```

In the above program, the values for x, y and z are taken from keyboard. Let it be 3 4 5. After execution of the program,

The 3 integers are 3 4 5

are stored in the file ex1.dat. If the file ex1.dat does not exist, it is created and the information is stored. If the file ex1.dat already exists, the information in it is cleared and the new information is stored.

Sample program using file concept to store the result as well as to read the data

```
#include<iostream.h>
#include<fstream.h>
void main()
{
int x,y,z;
ifstream fin;
ofstream fout;
fin.open(ex1.dat);
fout.open("ex2.dat");
fin>>x>>y>>z;
fout<<"The 3 integers are: ";
fout<<x<<"  "y<<" "<<z;
fout.close();
return;
}
```

infile.dat
(Not changed by the program)

```
3
4
5
```

outfile.dat
(After the program is run)

```
The 3 integers are 3 4 5
```

Note: There is no output to the screen and no input from the keyboard.

The streams fin, fout discussed and predefined streams cin and cout are objects. An object is a variable that has functions as well as data associated with it. fin, fout both have function named "open" associated with them. Two sample calls of these functions, along with the declaration of objects is as follows:

```
ifstream fin;
ofstream fout;
fin.open(f1);
fout.open(f2);
```

The function named open associated with "fin" is a different from the function named open associated with object "fout". One function opens a file for input and other opens a file for output. The compiler determines to which object the open function belongs to, by looking at the name of the object that precedes the dot operator. If two objects are of the same type, they may have different values, but uses the same member function.

Eg:

```
ifstream fin, fin1;
ofstream fout, fout1;
```

A datatype, whose variables are objects are called class. Since the member functions of an object are determined by its class, these functions are called as member functions of the class.

Here, as the "open" function of ifstream is different from the "open" function of ofstream. Similarly, both ifstream and ofstream has "close" as their member function, but they close the files in different ways.

4.1.3 To Check For Successful Opening Of The File

A call to the open function can be unsuccessful for a number of reasons. If we open an input file with an external file name and if there is no file

with that name then the open function fails. When this function fails we may not get an error message and the program will do some unexpected work. Hence we should test to see whether the call to open was successful and if it is not successful the program has to end. There is a member function called fail() for each of the classes “ifstream” and “ofstream” . The “fail” function takes no arguments and returns a Boolean value. This Boolean value can be used in a repetitive or conditional statement.

A call to fail() should be placed immediately after each call to open. If the call to open fails when the function returns true otherwise it returns false. Whenever an opening fails an exit function can be called using the header file < stdlib.h>. When the exit statement is executed, the program ends immediately. Any integer value may be used with the exit statement.

Syntax:

exit(integer value);

When the exit statement will be executed the program ends immediately. Any integer value will be used as argument, but generally “1” is used for call to exit i.e., caused by an error and “0” is used in other cases.

e.g:
```
ifstream fin;
fin.open(“f1.dat”);
if fin.fail()
{
cout<<”error while file opening”;
exit(1);
}
```

We can declare the input and output file names as variables in character array. These variables can be used to read the file names as string data.

Ex: Program to store some integers in a file "infile.dat" and the result is stored in another file "outfile.dat".

```
#include<iostream.h>
#include<fstream.h>
#include<stdlib.h>
int main()
{
int fact(int x);
char f1[20],f2[20];
ifstream fin;
ofstream fout;
cout<<"enter i/p file name";
cin >>f1;
```

```
cout<<"enter o/p file name";
cin >>f2;
fin.open(f1);
if(fin.fail())
{
cout<<"error while i/p file opening";
exit(1);
}
fout.open(f2);
if(fout.fail())
{
cout<<"error while o/p file opening";
exit(1);
}
fout<<"value \t factorial \n";
int x;
while(fin>>x)                    //satisfied if there is a next number to read
{
fout<<x<<"\t"<<fact(x)<<"\n";
}
fin.close();
fout.close();

return 0;
}
int fact(int x)
{
int f=1, I=1;
while(I<=x)
{
f=f*I;
I++;
}
return f;
}
```

output:

enter i/p file name infile.dat
enter o/p file name outfile.dat

infile.dat

5
6

outfile.dat

value	factorial
5	120
6	720

Before executing the above program, a new file (any name can be given) "infile.dat" is created and the values whose factorials to be calculated are entered and saved. When the above program is executed, the system prompts to enter the input file name. The filename used to store the input data is entered. Here it is "infile.dat" because the input data is stored here. Next, the system prompts for an output file name. An output filename has to be entered. Here, "outfile.dat" is the output file. The data from the file infile.dat is read, using the statement,

"fin>>x"

in the while loop. "fin>>x" reads the next number stored in the inputfile and the while loop is executed till there is next number to read in the file.

Note: A stream can be an argument to a function. The formal parameter passed must be a call by reference stream parameter. Depending upon the number of files used, stream parameters can be passed.

4.2 Character I/O

C++ allows to read and display character values. The character values are stored in character type variables. The extraction operator can be used to read a character of input but when extraction operator is used to skipping of blank, new line etc., are done automatically.

4.2.1 Member functions get(),put() and putback()

The member functions get() and put() can be used to perform character input and output. Every input stream(input-file stream or cin) has a member function *get* , which can be used to read one character of input. get() reads the next input character no matter if it is a blank or a newline character. The get() takes one argument which should be of type character. When the get() is called the next input character is read and passed into the argument variable.

Syntax:

Inputstream.get(char vari);

Suppose the program code is as follows:

```
char c1,c2,c3;
```

```
cin.get(c1);
cin.get(c2);
cin.get(c3);
```
and the following lines of input are read:
```
XY
ZA
```

then c1 is set to 'X', c2 is set to 'Y', c3 is set to '\n' as we press "enter" after writing the first line of input.

//Program to demonstrate get(),put() and putback()

```
#include<iostream.h>
#include<ctype.h>
void main()
{
char next;
do
{
cin.get(next);
cout.put(next);
}
while(next!='\n');
cin.putback(next);
cin.get(next);
cout.put(next);
}
```

output:

```
hello
hello
```

To read data from a file, input file stream object is placed instead of cin. Every output stream has a member function named put(). The member function put() takes one argument which should be an expression of type character. When a member function put() is called the value of its argument is output to the output stream.

Syntax:

```
outstream.put(char expression);
```

e.g.:

```
cout.put(next);
cout.put('a');
```

Note: The function putback() is a member of input stream. It takes one argument of type character and places it back into the input stream. The argument can be any expression that evaluates to a value of type character.

Syntax: Inputstream.putback(char exp);
e.g: cin.putback(next);

4.2.2 The eof() member function

Every input-file stream has a member function called eof(),that can be used to determine when all of the file is read and there is no more input left for the program. The eof() works best for the input that is text. If *fin* is a input file stream , then call to the function is written by

fin.eof().

This is a Boolean expression that can be used to control a while loop, **do-while** or **if-else** statement. This expression is *true* if the program has read past the end of input file, otherwise the above expression is not satisfied. Generally the call to this function is done with (!) symbol, as we test that we are not at the end of file.
Eg:
Suppose fin is the input file stream, then the entire contents of the file can be written to the screen with the following while-loop:

```
fin.get(next);
while(!fin.eof())
{
cout<<next;
fin.get(next);
}
```

4.2.3 Predefined Character Functions

Some predefined character functions are put in <ctype.h> header file. So, the functions in ctype.h can be used by including the include directive.

#include<ctype.h>

Some of the predefined functions in ctype.h are:

int toupper (char expr);
It returns the uppercase version of the character expression.

e.g.:

```
char c=toupper('a')
cout<<c;
Or
cout<< char(toupper('a'));
```

int tolower(char expr);
It returns the lowercase version of the character expression.

int isupper(char expr);
It returns true if the char expr is in upper case otherwise it returns false.

e.g.:

```
char c;
c='a';
if(isupper(c)
cout<< "is upper case";
else
cout<<"is lower case.";
```

int islower(char exp);
Returns true if character exp is a lower case letter otherwise returns false.

e.g:

```
char c;
c = 'a';
If(islower(c)
cout << c << " is a lower case.";
```

int isalpha (char exp);
Returns true if the character exp is a letter of alphabet otherwise returns false.

e.g.:

```
char c;
c='$';
if(isalpha(c)
cout << "is a letter";
else
cout << c<< "is not a letter";
```

int isdigit (char exp);

Returns true if the character exp is one of the digits from '0' to '9' otherwise returns false.

e.g.:
```
char c;
c='3';
If (isdigit(c)
cout<<c<< "is a digit";
```

int isspace(char exp);
It returns true if the character expression is a wide space such as blank space, \n, \t etc otherwise it returns false.

e.g.:
```
do
 {
cin.get(c);
cout.put(c);
}
while(!isspace(c);
```

4.2.4 Inheritance

When one class was derived from another class, the derived class was obtained from the previous class by adding new features.

Ex: The class of input file stream is derived from the class of all input streams by adding additional member functions such as open(), close(), fail(), eof(). The stream cin belongs to the class of all inputstream, but does not belong to the class of input file streams because cin has no member functions open(), close() etc.

If " **istream**" is used as the type for an input stream parameter then the argument corresponding to that formal parameter can be either the stream cin or an input file stream of type ifstream.
If "**ostream** " is used as the type of an output stream parameter then the argument corresponding to that formal parameter can be either the stream cout or an output file stream of type of stream.

```
#include<iostream.h>
#include<fstream.h>
void add(istream &in);
void main()
{
ifstream fin;
fin.open("file1.dat");
```

```
cout<<"data from file\n";
add(fin);
cout<<"enter 2 integers :";
add(cin);
fin.close();
}
void add(istream&in)
{int n1,n2;
in>>n1>>n2;
cout<<"sum of" <<n1<<" and " << n2 << " is  " << n1+n2 <<"\n";
}
```

output:
data from file
sum of1and2is3
enter 2 integers :4
5
sum of 4 and 5 is 9

Note: The function parameters can have default arguments that provide values for the parameters, when the corresponding argument is omitted in the call.

4.3 Formatting Output

The layout of a programs output is called the format of the output. C++ supports a number of features that could be used for formatting the output. These features include:

- Manipulators
- Ios class functions and flags.

Manipulators are operators used to format the data display. Some of the manipulators are endl, setw, set precision etc. In order to use a manipulator the header file<iomanip.h> must be included in the program. The manipulators are used with insertion operator. The “endl” manipulator, when used in output statement causes a line feed to be inserted. It works like new line character “\n”. For every manipulator there is an equivalent “ios” class function. The ios class contains a large number of member functions that would help to format the output. These

can be used to format the screen as well as format the files. To format the file the respective ofstream object should be attached with the functions.

Formatting with manipulators and class ios functions:
(Tools for i-o stream)

Manipulators	**Equivalent ios class** function
setw()	width();
eg:	eg:
float m=15	cout.width(7)
setw(m);	cout<<m.
cout<<m;	

It specifies a common field width for all the numbers and make them print right justified.

Setprecision()	precision()

It specifies the number of digits to be displayed after the decimal point. The setting is retained until is changed or reset. The output will be rounded.

e.g.:	Eg:	
cout << setpricision(2)<<5.678	cout.precision(2);	cout<<5.678;

setiosflags(ios::fixed) setf(ios::fixed, ios::floatfield)
It shows output in fixed floating point.

setiosflags(ios::showpoint) setf(ios::showpoint)

It shows decimal point in following point is floating point Output.

```
//program using manipulators
#include<iostream.h>
#include<iomanip.h>
int main()
{
double num = 67857.765;
cout<<setiosflags(ios::fixed)
<<setiosflags(ios::showpoint)
<<setprecision(1)
<<setw(12)
<<num<<"\n";
return 0;
```

```
}
output:
  67857.8
```

//program using iosclass functions

```
#include<iostream.h>
#include<iomanip.h>
int main()
{
double num = 67857.765;
cout.setf(ios::fixed);
cout.setf(ios::showpoint);       //each function should be called
cout.precision(1);               //seperately
cout.width(12);
cout.precision(1);
cout<<num<<"\n";
return 0;
}
```

output:
67857.8

setiosflags(ios::left) setf(ios::left,ios::adjust field)
It left justifies the output.

setiosflags(ios::right) setf(ios::right,ios::adjust field)
It right justifies the output.

setiosflags ios:showpos) setf(ios::showpos)
It prefix + before positive integers.

setiosflags(ios:internal) setf(ios::internal,ios::adjust field)
It is used for padding after sign.
e.g.: + 65.74

setiosflags(ios::scientific) setf(ios::scientific,ios::float field)
It shows the o/p in exponent form.

Resetiosflags(ios::internal) unsetf(ios::internal)
The flags can be reset using the above function. The respective argument is passed to reset the option. Here the internal flag is reset. Similarly the other flags like left, fixed etc., can also be reset.

Setfill(int C); fill(int c)

It fills the field with a particular character.

4.4 Working With Files

A **file** is a collection of related data stored in a particular area of the disk. We have already studied that the I/O system of C++ handles file operation by using file streams as an interface between the programs and the files. The stream that supplies data to the program (reads from the file) is known as input stream and the stream that receives data from the program (writes to the file) is known as the output stream.

The i/p operation creates an input stream and links it with the program and the input file. The o/p operation establishes an o/p stream and links it with the program and the o/p file.

The ifstream, ofstream and fstream are the classes that define file handling methods. These are derived from fstreambase and also i/ostream. These are defined in fstream. So, fstream should be included in this file.

Class	**Content**
fstreambase	Provides operations common to the file streams. It serves as base for ifstream, ofstream, stream classes. Contains open() and close() functions.
ifstream	Provides i/p operations. Contains open() with default input mode. Inherits get(), getline(),read(),tellg(),seekg() functions from istream.
ofstream	Provides output operations. Contains open() with default output mode. Inherits put(),seekp(),tellp(),write() from ostream.
fstream	Provides support for simultaneous i/o operations. Contains open() with default i/p mode. Inherits all the functions from istream and ostream classes through iostream.

4.4.1 File Modes

open() function is used to create new files as well as to open existing files. These functions can take two arguments. The general form of function open() with two arguments is:

stream-object.open("filename",mode);

The second argument mode specifies the purpose for which the file is opened. The file modes can take one or more of the following parameters.

Parameter	Meaning
ios::app	Append to end-of-file.
ios::ate	Go to end-of-file on opening.
ios::binary	Binary file.
ios::in	Open file for reading only.
ios::nocreate	Opens fails if the file does not exist.
ios::noreplace	Opens files if files already exists.
ios::out	Open file for writing only.
ios::trunc	Delete the contents of the file if it already exists.

- Opening a file in ios::out mode opens it in ios::trunc mode by default.
- Using both ios::app and ios::ate , opens the file and pointer points to the end of the file. ios::app allows to add data to the end of the file only. Ios::ate permits to add or modify the existing data anywhere in the file. In both cases, a file is created if it does not exist.
- Ios::app can be used only with files capable of output.
- Creating a stream using ifstream, means input and creating using ofstream, means output.
- Fstream class does not provide a mode by default. So, the mode parameters are provided.
- The mode can combine two or more parameters using bit wise OR operator.

Ex: fout.open("data",ios::app| ios::nocreate)

4.4.2 File Pointers And Manipulators

Each file has two associated pointers known as file pointers. One of them is called input or get pointer and the other is called output or put pointer. These pointers are used to move through the files while reading or writing. The input pointer is used for reading the contents of a given file location and the output pointer is used for writing to a given file location.

- When a file is opened in i/p(read) mode, the i/p pointer is automatically set at the beginning and the file is read from the start.
- When a file is opened in o/p(write) mode, the existing contents are deleted and the o/p pointer is automatically set at the beginning to write.
- To open a file in the existing mode and add more data, the file is opened in append mode and the o/p pointer is placed at the end of the existing file for writing purpose.

4.4.3 Functions For Manipulation Of File Pointers

The file stream classes support the following functions to manage file pointers:

- Seekg() moves i/p(read or get) pointer to a specified location.
- Seekp() moves o/p(write or put) pointer to a specified location.
- Tellg() gives the current position of the get pointer.
- Tellp() gives the current position of the put pointer.

Ex: infile.seekg(10) moves the file pointer to the byte number 10. The byte number starts from zero. That is, it will point to byte number 11.

Ex: To find the size of a given file:

```
ofstream fout;
fout.open("hello",ios::app);
cout<<fout.tellp();
```

The o/p pointer is moved to the end of the file and tellp() gives the number of bytes in the file.

Format:
seekg(offset,refpos); //Moves the files get pointer
seekp(offset,refpos);//Moves the files put pointer

The parameter offset represents the number of bytes to be moved from the location specified by the parameter refpos. The refpos can take one of the following:

- ios::beg start of the file
- ios::cur current position of the pointer
- ios::end end of the file

e.g.:

fout.seekg(0,ios::beg)	Go to start
fout.seekg(0,ios::cur)	Stay at the current position
fout.seekg(0,ios::end)	Go to end of file
fout.seekg(m,ios::beg)	Move to (m+1)th byte in the file
fout.seekg(m,ios::cur)	Move forward m bytes from the current position
Fout.seekg(-m,ios::cur)	Move backward m bytes from the current position
Fout.seekg(-m,ios::end)	Move backward m bytes from the end position

4.4.4 Sequential Input And Output Operations

Sequential access of files is to access the contents of the file from the beginning one after the other.

Ex: Accessing in tapes and disks. The operations that are performed on the file sequentially are reading and writing.

File streams support a number of member functions for performing the i/p and o/p operations on files.
put() and get()functions are designed for handling a single character at a time.

Ex:
```
fout.put(ch);
fin.get(ch);
```

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

Formats:

```
Infile.read((char *) &V, sizeof(V));
Outfile.write((char *) &V,sizeof(V));
```

These functions take two arguments. First is the address of the variable V and the second is the length of that variable in bytes. The address of the variable has to be type cast to char*.

```
#include<iostream.h>
#include<fstream.h>
#include<iomanip.h>
class invent
{
char name[20];
int code;
float cost;
public:
void readd();
void writed();
};
void invent:: readd()
{
cout<<"Enter name:";
cin>>name;
cout<<"Enter code:";
cin>>code;
cout<<"Enter cost:";
cin>>cost;
}
void invent::writed()
{
cout<<”Name:”<<name<<"\n"<<”Code:”<<code;
cout<<”\n"<<”Cost:”<<cost<<endl;
}
void main()
{
int i;
invent item[5],item1;
fstream fil;
fil.open("STOCK.DAT",ios::in|ios::out);
cout<<”Enter the details of 3 items\n”;
for(i = 0;i < 3; i++)
{
item[i].readd();
fil.write((char*)& item[i],sizeof(item[i]));
```

```
}
fil.seekp(sizeof(item1));
item1.readd();
fil.write((char*)& item1,sizeof(item1));
fil.seekg(0);
for(i=0;i<3;i++)
{
fil.read((char*)& item[i],sizeof(item[i]));
item[i].writed();}
fil.close();
}
```

Output:

```
Enter the details of 3 items
Enter name:c++
Enter code:1
Enter cost:123
Enter name:Datastructures
Enter code:2
Enter cost:345
Enter name:DataMining
Enter code:3
Enter cost:432
Name:C++
Code:1
Cost:123
Name:Datastructures
Code:2
Cost:345
Name:DataMining
Code:3
Cost:432
```

4.4.5 Updating Random Access Files

Random access of files is the process of accessing a file randomly from any location. Ex: Access from disk files. The operations that are performed randomly are

- Read
- Write
- Update

- ✓ Display
- ✓ Modify an existing item
- ✓ Adding a new item
- ✓ Deleting an existing item

The object length can be found using sizeof() operator. The location of the desired (say m th) object can be found by multiplying object size with m. The file pointer is set to that object by using seekg() or seekp(). To find the total number of objects in a file is found by dividing filesize with the objectsize. The filesize can be found using the function tellp() or tellg().

Program to demonstrate the random access file updating

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

class invent
{
char name[10];
int code;
float cost;
public:
void readd();
void writed();
};
void invent:: readd()
{
cout<<"Enter name:";
cin>>name;
cout<<"Enter code:";
cin>>code;
cout<<"Enter cost:";
cin>>cost;
}
void invent::writed()
{
cout<<name<<"\t"<<code<<"\t"<<cost<<endl;
}
void main()
{
invent item;
fstream fil;
```

```
fil.open("STOCK1.DAT",ios::in|ios::out|ios::ate);
fil.seekg(0,ios::beg);
cout<<"Current contents of stock\n";
while(fil.read((char*)&item,sizeof(item)))
{item.writed();
}
fil.clear();

cout<<"ADD AN ITEM\n";
item.readd();
char ch;
cin.get(ch);
fil.write((char *)&item,sizeof(item));
fil.seekg(0);
cout<<"Contents of appended file\n";
while(fil.read((char *) &item,sizeof(item)))
item.writed();

int last = fil.tellg();
int n = last/sizeof(item);

cout<<"Number of objects="<<n<<endl;
cout<<"Total bytes="<<last<<endl;
cout<<"Enter object no to be updated:"<<endl;
int object;
cin>>object;
cin.get(ch);
int loc = (object-1) * sizeof(item);
if(fil.eof())
fil.clear();
fil.seekp(loc);

cout<<"Enter new values\n";
item.readd();
cin.get(ch);
fil.write((char*)&item,sizeof(item))<<flush;

fil.seekg(0);
cout<<"Contents of updated file\n";
while(fil.read((char *)&item,sizeof(item)))
{item.writed();}
```

```
fil.close();
return ;
}
```
output:
```
Enter code:1
Enter cost:123
Contents of updated file
c       1       123
Current contents of stock
c       1       123
ADD AN ITEM
Enter name:java
Enter code:2
Enter cost:432
Contents of appended file
c       1       123
java    2       432
Number of objects=2
Total bytes=32
Enter object no to be updated:
1
Enter new values
Enter name:c++
Enter code:1
Enter cost:123
Contents of updated file
c++     1       123
java    2       432
```

4.5 Summary

- We have covered about the streams, the advantages of using files and about file input and output.
- We have studied the different character member functions like get(),put(), and putback(). Also we covered the usage of eof() member function.

- We have studied in detail about different stream functions and manipulators for formatting the output.

- We have studied in detail the working of file, file modes, filepointers, file manipulation functions. The details of writing sequential files and random access files are covered.

4.6 Model Questions

File mode: It specifies the purpose for which the file is opened.

File pointer: The input and output file pointers are to move around the file.

Stream: A flow of characters.

4.7 Model Questions

1. Explain in detail about file I/O.
2. Explain the usage of put(),get() and putback() functions.
3. Give some of the predefined character functions and explain their usage.
4. Write in detail the working of files.
5. Explain the tools for formatting the output.

4.8 References

Object-oriented programming with C++,
by E. Bala Gurusamy.

Problem solving with C++ **by Walter Savitch**

AUTHOR:

M. NIRUPAMA BHAT, MCA., M.Phil.,
Lecturer,
Dept. Of Computer Science,
JKC College, Guntur.

Lesson 5: Structures And Classes

Objectives

After completing this lesson you will understand:

- About structures, initializing the structures and passing structures as arguments into functions.
- To define a class and know about the access specifier: public and private, defining a member function.
- To learn the difference between the structure and the class
- To study in detail about the constructors.

Structure Of The Lesson

5.1 Structures

Structures combine logically related data items into a single unit. The data enclosed within a structure are known as members and they can be of same type or different type. It is viewed as heterogeneous user-defined data type.

5.1.1 Defining Structure

A structure is a collection of variables of different data types grouped under a common name. The syntax of structure is as follows:

```
struct  tag name
{
data type1 member vari1
data type 2 member vari2
:
:
} ;
```

The keyword **struct** announces that it is a structure type definition. The tag name gives the name of the structure. The identifiers inside the braces are the member variables or of member names. The structure should end with a semi colon (;). Structure definition can be placed inside the main or before main function. After defining the structure definition, it can be used just like a predefined data type.

```
struct account
{
double balance;
double irate;
float term;
};
account acc1,acc2;
```

The structure variables can hold the values like any other variables. The structure value is a collection of smaller values called the member values. The smaller variables are called member variables.

The member variables can be accessed by using the structure variable followed by a dot(.), and then a member variable. Thus, the dot operator is used to specify a member variable of a structure variable.
Syntax: structure variable name. Member variable

In the above example the structure account has **three** member variables: balance, irate, term. acc1 and acc2 are structure variables of type account. Then the member variables of acc1 are:

acc1.balance
acc1.irate
acc1.term
Similarly acc2 has the member variables.
acc2.balance
acc2.irate
acc2.term

The first two variables are of type double and the third variable is of type float. The member variables can be assigned values using assignment statements. They can be used like any other variable.

```
e.g.:   acc1.balance = 1000.0;        //initialization using assignment
        acc1.irate = 2.0;
        acc1.term=2.0;
        acc1.balance = acc1.balance + interest;
        // arithmetic operation using structure variables.
```

The member variables can also be initialized during the declaration of the structure variable. To give a structure variable a value, it is followed by an equal sign and a list of member values enclosed within braces.

```
e.g.:   struct date
        {
        int month;
        int day;
        int year;
        };
```

Once the type date is defined, a structure variable birthday can be initialized as follows:

```
date birthday = { 12,3,1969};
```

During initialization, the order of the initializing values should match with the order of the member variables.

Then, birthday.month receives the value 12, birthday.day receives the value 3 and birthday.year is initialized with the value 1969. The number of member variables and the number of initializing values should be same.

Two or more structure types may use the same member names. The dot operator and the structure variable specify to which structure the member variable belongs.

eg:

```
struct fertilizerstock
{       double quantity;
        double nitrogencontent;
};
struct cropyield
{       int quantity;
        double size;  };
fertilizerstock  super;
cropyield apples,oranges;
```

fertilizerstock and **cropyield** are two structures types. Super and apples are variables of fertilizerstock and cropyield respectively. The quantity of super fertilizer is stored in **super.quantity** and the quantity of apples are stored in **apples.quantity.**

The structure value can be viewed as a collection of member values or a single variable. The structure value of one structure variable can be assigned to another structure variable using = sign. Thus a structure variable can be assigned into another structure variable of the same type by using an assignment operator.

If apples and oranges are two variables of type **cropyield**. Then

```
apples = oranges
```

is equivalent to

```
apples.quantity = oranges.quantity;
apples.size = oranges.size;
```

5.1.2 Initializing Structures

A structure can be initialized at the time of declaration. A structure variable can be given a value, by giving an equal sign and a list of member values enclosed in braces.

Eg:

```
struct date
{
        int month;
        int day;
        int year;
};
date duedate = { 12,31,1999};
```

The initializing values should be given in order with the member variables in the structure definition. If there are fewer initializing values than struct members, the provided values are initialized data members, in order. The remaining data members without initial values are initialized to a zero value of an appropriate type of the variable.

5.1.3 Structures As Function Arguments

Structures can be passed as arguments into functions. They can be passed as call by value or call by reference arguments.

Eg: A program to input your birth date and to display it.

```
#include<iostream.h>
struct date                 //structure definition
{
int day;
int month;
int year;
};
void getdate(date&);   //function prototypes
void putdate(date);
void main()
{
date bday;
getdate(bday);
putdate(bday);
return;
}
void getdate(date &bday)
{
cout<<"Day";
cin>> bday.day;
cout<<"Month";
cin>>bday.month;
```

```
cout<<"Year";
cin>>bday.year;
}
void putdate (date bday)
{
cout<<"My birth day is on";
cout<<bday.day<<" "<<bday.month<<" "<<bday.year;
}
```

output:

```
Day3
Month12
Year69
My birth day is on3 12 69
```

Program to input your birth date and to display it in another way.

```
#include<iostream.h>
#include<string.h>
struct date                    //structure definition
{
int day;
char month[15];
int year;
};

date getdate(int,char[],int);
void putdate(date);
void main()
{
date bday;
bday =getdate(15,"aug",1985);
putdate(bday);
}
date getdate(int x,char y[],int z)
{
date temp;
temp.day =x;
strcpy(temp.month,y);
temp.year=z;
return temp;
}
void putdate (date bday)
{
```

```
cout <<"my birth day is on";
cout<<bday.day<<" "<<bday.month<<" "<<bday.year;
}
```

output:

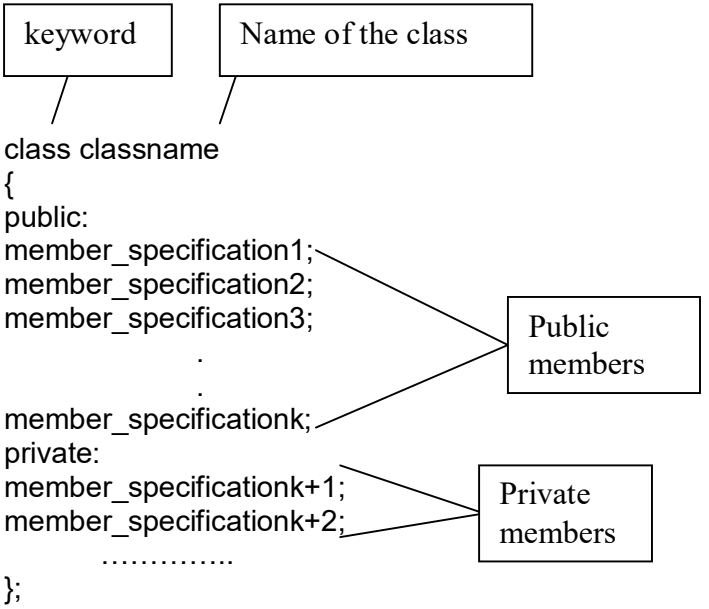
My birth day is on15 aug 1985

5.2 Classes

Classes are the basic language constructs of C++ for creating user defined data types. They are syntactically an extension of structures. Class follows the principle that the information about a module should be private to the module unless it is specifically declared public.

5.2.1 Defining A Class

A class is a data type whose variables are objects. Object is a variable that has data values as well as member functions. Classes are created using the keyword “class”. A class declaration defines a new data type that links the code and the data. This data type is used to declare the objects of that class. Hence class is a logical abstraction and an object is a physical abstraction. The syntax of the class is



The class definition contains the data members and can have prototypes of its member functions. The definition of the member function is given elsewhere, the class definition ends with a semicolon.

A sample class definition is given in the program below.

Program to demonstrate an example of classes

```
#include<iostream.h>
class dayofyear
{
public:
void output();//          <-------  member function prototype
int month;
int day;
};

int main()
{
dayofyear       today,birthday;
cout<<"Enter today's date\n"<<"Enter month as number:";
cin>>today.month;
cout<<"Enter day of the month:";
cin>>today.day;
cout<<"Enter your birthdate:\n"<<"Enter month as
number:";
cin>>birthday.month;
cout<<"Enter day of the month:";
cin>>birthday.day;
cout<<"Today's date is ";
today.output();
cout<<"Your birthday is ";
birthday.output();
if (today.month == birthday.month && today.day ==
birthday.day)
cout<<"Happy Birthday!\n";
else
cout<<"Good day!";
return 0;
}
void dayofyear::output()
{
        cout<<"month ="<<month<<",day = "<<day<<endl;
```

```
}
```

output:

```
Day3
Month12
Year69
my birth day is on3 12 69my birth day is on15 aug 1985Enter today's date
Enter month as number:12
Enter day of the month:3
Enter your birthdate:
Enter month as number:1
Enter day of the month:3
Today's date is month =12,day = 3
Your birthday is month =1,day = 3
Good day!
```

The type dayofyear defined is a class definition for objects whose values are dates. The member variables month and day stores the month and day of the year in integers. There is a member function called output, whose prototype is given in the definition. A class definition may contain the function prototype or the function definition. Two objects called today and birthday of data type day of year are declared. The member function output can be called with the object today or birthday as:

```
today.output();
birthday.output();
```

When a member function is defined the definition must include class name because there may be two or more classes that have member functions with the same class name. The member function is defined in the same way as any other function except the class name and the scope resolution operator are given in the function heading. The (.) dot operator and (::) scope resolution operators are used to tell which class the member belongs to. The scope resolution operator is used with class name, whereas the dot operator is used with the objects. The following function call will output the data values stored in the object today:

```
today.output();
```

The scope resolution operator specifies that the function output() belongs to the class dayofyear.

The class name that precedes the scope resolution operator is known as type qualifier because it specializes the function to one type of class. The member function is defined the same way as any other function except that the class name and scope resolution operator (::) are given in the function heading. The syntax of member functions is as follows.

Syntax:

```
Returntype   classname  :: function name(parameter list)
{
function body;
}
eg:
//uses iostream.h
void dayofyear::output()
{
        cout<<"month ="<<month<<",day = "<<endl;
}
```

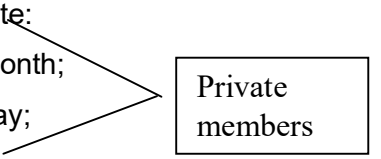
5.2.2 Public And Private Members

All the member variables and member functions, which are listed after keyword private in the class definition, are called private members of the class. When a variable is defined as private, it cannot be directly accessed in the program except within the definition of the member function. If it is accessed from the main program, an error message is given. When they are declared as public in the class definition, they can be accessed anywhere throughout the program. If private or public are not declared the compiler will take a default access specifier ie., private.

Program to demonstrate a simple example of a class

```
#include<iostream.h>
class dayofyear
{
public:
        void input();
        void output();//<------ member function prototype
        void set(int newmonth, int newday);
//Precondition:newmonth and newday take integer values
of //month and day of the date
//Postcondition:The data is reset according to the
arguments
int getmonth();
```

```
//Returns the month of its corresponding date object.
int getday();
//Returns the day of the month
private:
int month;
int day;
};
```




```
int main()
{
dayofyear       today,birthday;
cout<<"Enter today's date\n";
today.input();
cout<<"Todays date is :";
today.output();
birthday.set(3,21);
cout<<"Birthday:";
birthday.output();
if     (today.getmonth()     ==     birthday.getmonth()     &&
today.getday() == birthday.getday())
cout<<"Happy Birthday!\n";
else
cout<<"Good day!";
return 0;
}
//Memberfunction definitions
void dayofyear::input()
{
cout<<"Enter the month as a number:";
cin>>month;
cout<<"Enter the day of the month:";
cin>>day;
}
void dayofyear::output()
{
        cout<<"month ="<<month<<",day = "<<day<<endl;
}

void dayofyear::set(int newmonth, int newday)
{
month = newmonth;
day = newday;
}
```

```
int dayofyear::getmonth()
{
return month;
}

int dayofyear::getday()
{
return day;
}
```

output:

```
Enter today's date
Enter the month as a number:1
Enter the day of the month:2
Todays date is :month =1,day = 2
Birthday:month =3,day = 21
Good day!
```

To assign one object to another object we can use an assignment operator. The member variables of the source object, is copied into the target object.

```
birthday = today;
```

It works as follows:

```
birthday.month = today.month;
birthday.day = today.day;
```

5.2.4 Accessor Functions

Accessor functions are public functions of a class, which returns the private variables of the class. These functions provide some kind of access to the private variables and hence are called accessor functions.

e.g:

```
int dayofyear::getmonth()
{
return month;
}

int dayof year::getday()
{
return day;
}
```

As month and day cannot be directly accessed from the main program a function to return these values are written.
e.g: //This is illegal as month and day are private variables

```
if (today.month == birthday.month && today.day == birthday.day)
        cout<<"Happy Birthday!\n";
        else
        cout<<"Good day!";
        This can be modified to:
        //This is legal
if (today.getmonth() == birthday.getmonth() && today.getday() == birthday.getday())
        cout<<"Happy Birthday!\n";
        else
        cout<<"Good day!";
}
```

5.2.5 Difference Between Structures And Classes

Structures are normally used with all the member variables as public and with no member functions. A class contains member variables and member functions. The member variables and member functions can be declared as private or public. By default the members of structure are public and the members of a class are private. But at least one member function of the class should be defined as public function.

5.2.6 Properties Of A Class

- Classes have both member variables and member functions.
- A member of a class may be public or private.
- By default all the members of a class are labeled as private members.
- A private member of a class cannot be used elsewhere except within the definition of the member functions of the class. The name of the member functions for a class can be overloaded just like any other function.

- A class may use (any) another class type as its member variables.
- A function may have formal parameters of class type.
- A function may return an object of class type.

5.3 Constructors

Defining A Constructor : A constructor is a special type of member function defined in the class definition. It is used to initialize all or some of the member variables of the objects in the class.

A constructor is automatically called when an object of its associated class is declared.

A constructor must have the same name as the class name. A constructor definition cannot return any value. A constructor should be placed in the public section of the class definition.

The definition of a constructor can be given in the same way as a member function.

A constructor is declared and defined as follows:

```
// constructors defined outside the class
class integer
        {
        int m,n;
        public:
        integer();//default constructor declared
integer(int m,int n);//parameterized constructor ……..
        ……..
        };
integer::integer()        //constructor defined
        {
          m=0;n=0;
        }
integer::integer(int x, int y)
        {
         m = x;
          n = y;
        }
```

- The constructor can also be defined when it is declared in the class definition itself.

```
//class with constructor
class integer
        {
                int m,n;
        public:
        integer()
        {
         m=0;
          n=0;
        }
        integer(int x, int y)
        {
         m = x;
          n = y;
        }          ……..
        ……..
        };
```

5.3.1 Calling A Constructor

When a class contains a constructor like the one defined as above, the object created by the class is automatically initialized.

For example, the declaration, integer n1;//object n1 is created
not only creates the **object** int1 of type integer but also initializes its data members m and n to zero. There is no need to write any statement to invoke the constructor. The constructor is automatically called when the object is declared.

Constructor cannot be called like a member function.
eg: n1.integer(15,10);//illegal

When constructors are defined, an object cannot be declared with no arguments.The arguments must be added or a new constructor should be defined with no arguments.

5.3.2 Default Constructor

A constructor with no arguments is called default constructor. If no constructor is defined, the compiler will generate a default constructor.

If the user defines at least one constructor, then C++ will not generate any other constructor. If we do not want to initialize the default constructor with any values then an empty body can be defined.

```
integer()
{
}
```

Constructor can be called explicitly using an assignment operator.
e.g.: integer n1 = integer(10,5);//explicit call

Constructor can be called implicitly.
e.g.: integer int1(10,5);//implicit call

Constructor can be overloaded.In the above example, the constructor integer is overloaded by defining it once with no arguments and the other time by passing parameters. Thus it is overloaded.
e.g.:

```
class integer
{
int m,n;
public:
integer();//default constructor declared
integer(int m,int n);//parameterized constructor
        ……..
        ……..
};
```

Overloaded constructors

Write a program to add, sub, and divide rational numbers.

```
#include<iostream.h>
#include<stdlib.h>
class rno
{
int num,den;
public:
rno(int n, int d)
{
if (d == 0)
exit(1);
num=n;
den=d;
}
rno()
{
num=0 , den = 1;
}
```

```
rno(int n)
{
num = n;
den = 1;
}
void add(rno rn1,rno rn2)
{
num=rn2.den*rn1.num+rn1.den*rn2.num;
den=rn2.den*rn1.den;
}
void sub(rno rn1, rno rn2)
{
num=rn2.den*rn1.num-rn1.den*rn2.num;
den=rn2.den*rn1.den;
}
void mul(rno rn1,rno rn2)
{
num=rn2.num*rn1.num;
den=rn2.den*rn1.den;
}
void div(rno rn1, rno rn2)
{
num= rn1.num* rn2.den;
den=rn2.num*rn1.den;
}
void show()
{
cout<<num<<"/"<<den<<"\n";
}
};
void main()
{
rno r1(2,3);//constructor with 2 arguments is called
rno r2(3);//constructor with one argument is called
rno r3;//default constructor is invoked;
r3.add(r1,r2);
cout<<"Rational number 1:";
r1.show();
cout<<"Rational number 2:";
r2.show();
cout<<"Sum:";
r3.show();
cout<<"Subtraction:";
r3.sub(r1,r2);
r3.show();
```

```
cout<<"Multiplication:";
r3.mul(r1,r2);
r3.show();
cout<<"Division:";
r3.div(r1,r2);
r3.show();
}
```

output:

```
Rational number 1:2/3
Rational number 2:3/1
Sum:11/3
Subtraction:-7/3
Multiplication:6/3
Division:2/9
```

5.4 Destructors

A destructor is used to destroy the objects that have been created by the constructor. A destructor is a member function whose name is same as the class name but is preceded by a tilde. For example the destructor for the class **integer** can be defined as follows:

```
~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(or block or function as the case may be) to clean up the storage that is no longer accessible.

Note: Whenever memory is allocated dynamically using **new** operator in the constructors, delete operator should be used to free the memory. This is required because when the pointers go out of scope, a destructor is not called implicitly.

Program to demonstrate the destructors

```
#include<iostream.h>
int count=0;

class alpha
{
```

```
public:
alpha()//constructor
{
count++;
cout<<"\nnumber of object created"<<count;
}
~alpha()//destructor
{
cout<<"\nnumber of object destroyed"<<count;
count--;
}
};

int main()
{
        cout<<"\n\nenter main\n";
        alpha a1,a2,a3,a4;
        {
                cout<<"\n\nenter block1\n";
                alpha a5;
        }

        {
                cout<<"\n\nenter block2\n";
                alpha a6;
        }

        cout<<"\n\nre-enter main\n";

        return 0;
}
```

output:

```
enter main

number of object created1
number of object created2
number of object created3
number of object created4

enter block1

number of object created5
```

number of object destroyed5

enter block2

number of object created5
number of object destroyed5

re-enter main

number of object destroyed4
number of object destroyed3
number of object destroyed2
number of object destroyed1

5.5 Summary

- We have learnt how to define a structure and initialize the structure and to pass structure as parameter to a function.
- The definition of class, the public and private members of the class, how to assign an object to another object and accessor functions are covered.
- The properties of the classes and the difference between the structures and classes are learnt.
- The definition of constructor, the different types of constructors, and overloading of constructors are covered in detail.
- The destructors are also covered.

5.6 Technical Terms

Abstract data type: The data type in which the programmer do not have access to the details of how the values and operations are implemented.

Constructor: A special member function for automatically creating an instance of a class. This function has same name as the class.

Destructor: A function that is called to deallocate memory of objects of a class.

Data hiding: A property whereby the internal data structure of an object is hidden from rest of the program. The data can be accessed by the functions declared within the class.

Data member: A variable declared in the class.
Member function: A function declared within the class and not declared as friend. These functions can have access to data members and define operations that can be performed on the data.

Private member: A class member that is accessible only to the member and friend functions of the class.

Public member: A class member that is accessible to all the users of the class. The access is not restricted to member and friend functions. The public member of the base class can be easily inherited by the derived class.

5.7 Model Questions

1. What are the difference between classes and structures?
2. Explain the properties of classes.
3. What is a class? What is an object? Explain the definition of class, its access specifiers with an example?
4. What are accessor functions?
5. What are the different types of constructors? Explain with example.
6. What is overloading of constructor?
7. What is a default constructor?
8. What is a destructor?

5.8 References

Object-oriented programming with C++,

by E. Bala Gurusamy.

Problem solving with C++

by Walter Savitch

Mastering C++

by K.R.Venugopal, RajkumarBuyya, T.RaviShankar

AUTHOR:

M. NIRUPAMA BHAT, MCA., M.Phil.,
Lecturer,
Dept. Of Computer Science,
JKC College,
GUNTUR.

Lesson 6: Tools For Defining Abstract Data Types

Objectives

After completing this chapter you will understand about:

- Defining Abstract Data Type Operations on objects as non-member functions, i.e., friend functions
- Overloading different operators.
- How to compile the Abstract data types into libraries, so that it is used in the same way as predefined libraries.

Structure Of The Lesson

6.1 Defining ADT Operations

Till now ADT operations have been implemented as member functions of the class. For some operations it is better to implement it as ordinary functions. We define operations on objects as non-member functions. Here is a simple example.

Program to explain the accessor functions.

```
#include<iostream.h>
class dayofyear
      {
        public :
          dayofyear();              //constructor
          dayofyear(int m,int d); //constructor
          void read();
          void print();
int getmonth();          //accessor function
int getday();            //accessor function
       private :
         int month,day;
      };

dayofyear :: dayofyear()
 {
    month =1;
    day =1;
 }
dayofyear :: dayofyear(int m, int d)
 {
    month= m;
    day= d;
 }

void dayofyear ::read()
 {
   cout<< " Enter month and day";
   cin>>month>>day;
}
void dayofyear::print()
  {
     cout<<day<<'-'<<month;
}
int dayofyear::getmonth( )
```

```
{
    return month;
}

int dayofyear::getday( )
{
    return day;
}

//nonmember function
int equal (dayofyear d1, dayofyear d2)
 {
    if((d1.getmonth() == d2.getmonth()) && (d1.getday() ==
    d2.getday()))
    return 1;
else
   return 0;
}
int main( )
{
    dayofyear today, bach_birthday(12, 3);
    cout << "Enter today's date:\n";
    today.read();
    cout << "Today's date is ";
    today.print();
    cout << "Nirupama'sbirthday is ";
    bach_birthday.print( );
    if ( equal(today, bach_birthday))
        cout << "Happy Birthday \n";
    else
        cout << "Good day\n";
    return 0;
}
```

output:

```
Enter month and day8
3
Today's date is 3-8Nirupama'sbirthday is 3-12Good day
```

Here equal is not a member function. But, using the accessor functions, it compares the objects of type day of year. If “equal” is a member function, one of the dates are taken as calling object. To treat both the dates in the same way, it is better to make “equal “ an ordinary non-member function.

6.1.1 Friend Functions

A friend function of a class is a non-member function, which has access to private members of objects of that class. To make a function a friend of a class, the function prototype must be listed in the class definition. The prototype is preceded by the keyword "**friend**". Prototype may be placed in either private section or public section.

Syntax of a class definition with friend functions:

```
class class_name
{
 public :
          Member function declarations;
          friend prototype-for-friend-function1;
          friend prototype-for-friend-function2;
          ----------
          ----------
  private :
        Declaration of private members.
};

Eg :
        class dayofyear
        {
        public :
            friend int equal dayofyear d1, dayofyear d2;

private :
           int month, day;
        };

        int equal (dayofyear d1, dayofyear d2)
        {
          return( d1.month = = d2.month && d1.day = = d2.day);
        }
```

A friend function is not a member function. It is defined and called the same way as an ordinary function.

6.1.2 Difference Between Friend Functions And Member Functions

- A member function is defined with scope resolution operator and type qualifier whereas a friend function is defined like an ordinary function.
- Both function prototypes are written in the class definition but the keyword friend appears before the prototype declaration of friend function.
- A member function is always called with reference to an object whereas a friend function is called like a normal function.

  Note: Member functions are used if the function involves one argument.

  Eg: r3.add(r1,r2)
  r1.add(r2)

- Non-member functions are used if the function involves more than one argument of the same class type or different class type.

Program to explain the friend functions.

```
#include<iostream.h>
class dayofyear
      {
        public :
          dayofyear();
          dayofyear(int m,int d);
          void read();
          void print();
          friend int equal (dayofyear d1, dayofyear d2);
        private :
          int month,day;
      };

dayofyear :: dayofyear()
 {
     month = 1;
     day = 1;
}
```

```
dayofyear :: dayofyear(int m,int d)
{
    month= m;
     day = d;
 }
void dayofyear ::read()
 {
    cout<< " Enter month and day";
    cin>>month>>day;
}
void dayofyear::print()
  {
      cout<<day<<'-'<<month;
}
int equal (dayofyear d1, dayofyear d2)
 {
   if((d1.month == d2.month) && (d1.day == d2.day))
    return 1;
else
   return 0;
}
void main()
{
  dayofyear d1(10,20),d2(10,12),d3(9,5),d4(9,5);
if(equal (d1,d2))
  cout << "d1 & d2 are equal\n";
else
 cout<<"d1 and d2 are not equal\n ";
if (equal(d3,d4))
  cout << "d3 or d4 are equal\n";
else
 cout << "d3 and d4  are not equal\n";
}
```

Output:

```
d1 and d2 are not equal
 d3 or d4 are equal
```

A simple rule to use the member function or non-member function is

- To use a member function if the involved task uses only one object.
- To use a non-member function if the involved task uses two or more objects.

6.1.3 Constant Parameter Modifier

A "**const**" modifier is used with call by reference parameter within the function header. A reference parameter can be modified by a function. By using constant modifier, the reference parameter can be prevented from accidental modifications.

Syntax:

```
Returntype memberfunctionname
(const datatype1& s1,const datatype2& s2,…);
```

e.g. :

```
int equal (const dayofyear& d1, const dayofyear& d2)
 {
if((d1.month = = d2.month) && (d1.day = = d2.day)
return1;
else
return 0; }
```

Thus any modifications to the data in the objects can be prevented.

6.1.4 Constant Member Functions

We can declare a member function of class as a constant member function. This is done by writing "**const**" keyword at the end of the prototype and function header in the function definition. A constant member function does not change an objects data. Such a function may be an accessor function or it may simply print function.

Syntax:

```
//Class definition
class  Class_name
        {
          public :
             return_type member_function_name (parameter list)
const;
                    --------
         };
```

```
//Member function definition
        return_type class_name ::Function_name
        (parameter_list)const

        {
        function body
        }

        eg: class dayofyear
        {
        public :
      void print() const;
       --------
       --------
        };
void dayofyear :: print() const
 {
     cout << day << ‘  ‘ << month;
 }
```

6.1.5 Operator Overloading

Operator overloading is a mechanism of redefining the meaning of C++ operators, i.e., making an operator to behave differently at different instances. They can be member functions or friend functions**.** An operator such as +,-,*,/ etc is used on predefined data. In one way these operators are also functions. As functions can be overloaded, operators can also be overloaded. They can be directly applied on different objects. An operator definition is written in the same way as a function definition, except that the operation definition includes the reserved word "**operator**", before the operator name. The predefined operators such as +, -, etc. can be overloaded by giving them a new definition for a class type.

Syntax:

```
Return type class name :: operator symbol (Parameters list)
{
        body of the function

}
```

Eg: rno rno::operator + (rno r2);

To overload a + operator with a member function one argument has to be passed into the function and to overload “+” with a friend function, two arguments must be passed. The arguments that are passed can be reference arguments or value arguments.

Note: During operator overloading, the arguments are listed before and after the operator in the function call.

Eg: v3 = v1+v2;

With a function the arguments are listed after the function name.

v3= add(v1,v2);

Thus, the predefined operators can be overloading by giving them a new definition for the class type or object.

6.1.6 Rules For Operator Overloading

1. When overloading an operator at least one argument of the resulting overloaded operator must be class type.

2. An overloaded operator can be a friend of the class, but it is not compulsory. It can be a member of the class.

3. A new operator cannot be created, but an existing operator can be overloaded.

4. The number of arguments that an operator takes cannot be changed. For example: % is a binary operator. It cannot be overloaded into a unary operator and vice versa.

5. Precedence of operators cannot be changed.

The following operators can be overloaded.

.	class member operator
::	scope resolution operator
?:	conditional operator
.*	de-referenced member access
sizeof	size of operator

Program to apply arithmetic and relational operations by operator overloading

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

class ratio
{
    int p,q;
    public:
      ratio()
      {
        p=q=1;
      }
      ratio(int a,int b)
      {
        p=a;
        q=b;
        if(q<=0)
        {
            cout<<"improper argument";
            exit(0);
        }
        int m=p<q?p:q;
        for(int i=m;i>1;i--)
        {
            if((p%i==0) && (q%i==0))
            {
                p/=i;
                q/=i;
                break;
            }
        }
      }
      void get()
      {
        cout<<"enter input";
          cin>>p>>q;
        int m=p<q?p:q;

        for(int i=m;i>1;i--)
        if((p%i==0) && (q%i==0))
        {
            p/=i;
            q/=i;
```

```
            break;
          }
        }
        void put()
        {
          cout<<p<<"/"<<q<<"\n";
        }
        void put1()
        {
         cout<<p<<"/"<<q;
        }
friend ratio operator+(ratio &,ratio &);
friend ratio operator-(ratio &,ratio &);
friend ratio operator*(ratio &,ratio &);
friend ratio operator/(ratio &,ratio &);
void operator<(ratio &);
void operator<=(ratio &);
void operator>(ratio &);
void operator>=(ratio &);
void operator==(ratio &);
void operator!=(ratio &);
};
ratio operator+(ratio &r1,ratio &r2)
 {
        ratio temp;
        temp.p=(r1.p*r2.q)+(r2.p*r1.q);
        temp.q=r1.q*r2.q;
        int min=temp.p<temp.q?temp.p:temp.q;
        for(int i=min;i>1;i--)
        if((temp.p%i==0)&&(temp.q%i==0))
        {
            temp.p/=i;
            temp.q/=i;
            break;
        }
        return temp;
}
ratio operator-(ratio &r1,ratio &r2)
{
        ratio temp;
        temp.p=(r1.p*r2.q)-(r2.p*r1.q);
        temp.q=r1.q*r2.q;
        int min=temp.p<temp.q?temp.p:temp.q;
        for(int i=min;i>1;i--)
        if((temp.p%i==0)&&(temp.q%i==0))
```

```
        {
            temp.p/=i;
            temp.q/=i;
            break;
        }
        return temp;
}
ratio operator*(ratio &r1,ratio &r2)
{
         ratio temp;
        temp.p=r1.p*r2.p;
        temp.q=r1.q*r2.q;
        int min=temp.p<temp.q?temp.p:temp.q;
        for(int i=min;i>1;i--)
        {
          if((temp.p%i==0)&&(temp.q%i==0))
          {
            temp.p/=i;
            temp.q/=i;
            break;
          }
        }
        return temp;
}
ratio operator/(ratio &r1,ratio &r2)
{
        ratio temp;
        temp.p=r1.p*r2.q;
        temp.q=r2.p*r1.q;
        int min=temp.p<temp.q?temp.p:temp.q;
        for(int i=min;i>1;i--)
        {
          if((temp.p%i==0)&&(temp.q%i==0))
          {
            temp.p/=i;
            temp.q/=i;
            break;
          }
        }
        return temp;
}
void ratio::operator<(ratio &r)
{
        float f1,f2;
        f1=float(p)/float(q);
```

```
        f2=float(r.p)/float(r.q);
        if(f1<f2)
        {
            put1();
            cout<<"<";
            r.put1();
        }
        else
        {
            r.put1();
            cout<<"<";
            put1();
        }
}
void ratio::operator<=(ratio & r)
{
        float f1,f2;
        f1=(float)p/(float)q;
        f2=(float)r.p/(float)r.q;
        if(f1<=f2)
        {
            put1();
            cout<<"<=";
            r.put1();
        }
        else
   {
            r.put1();
            cout<<"<=";
            put1();
        }
}
void ratio::operator>(ratio & r)
{
        float f1,f2;
        f1=(float)p/(float)q;
        f2=(float)r.p/(float)r.q;
        if(f1>f2)
        {
            put1();
            cout<<">";
            r.put1();
        }
        else
        {
```

```
            r.put1();
            cout<<">";
            put1();
        }
}

void ratio::operator>=(ratio & r)
{
        float f1,f2;
        f1=(float)p/(float)q;
        f2=(float)r.p/(float)r.q;
        if(f1>=f2)
                    {
                                put1();
            cout<<">=";
            r.put1();
        }
        else
        {
            r.put1();
            cout<<">=";
            put1();
        }
}
void ratio::operator==(ratio & r)
{
        float f1,f2;
        f1=(float)p/(float)q;
        f2=(float)r.p/(float)r.q;
        if(f1==f2)
        {
            put1();
            cout<<"==";
            r.put1();
        }
        else
        {
            r.put1();
            cout<<"!=";
            put1();
        }
}
void ratio::operator!=(ratio & r)
{
```

```
        float f1,f2;
        f1=(float)p/(float)q;
        f2=(float)r.p/(float)r.q;
        if(f1!=f2)
        {
            put1();
            cout<<"!=";
            r.put1();
        }
        else
        {
            r.put1();
            cout<<"!=";
            put1();
        }
}
int main()
{
        ratio r1(25,15),r2,r3(1,1),r4(5,3);
                    r2.get();
        cout<<"r1=";
        r1.put();
        cout<<"r2=";
        r2.put();
        cout<<"r3=";
        r3.put();
        cout<<"r4=";
        r4.put();
        cout<<"addition of r1 r2 is ";
        r3=r1+r2;
        r3.put();
         cout<<"subtraction of r1 r2 is ";
        r3=r1-r2;
        r3.put();
        cout<<"multiplication of r1 r2 is ";
        r3=r1*r2;
        r3.put();
        cout<<"division of r1  r2 is ";
        r3=r1/r2;
        r3.put();
        r1<r2;
        cout<<"\n";
        r1>=r4;
        cout<<"\n";
        r2>r4;
```

```
        cout<<"\n";
        r1==r2;
        cout<<"\n";
        r1<=r2;
        cout<<"\n";
        r1==r4;
        return 0;
}
```

output:

```
        enter input2
        3
        r1=5/3
        r2=2/3
        r3=1/1
        r4=5/3
        addition of r1 r2 is 7/3
        subtraction of r1 r2 is 1/1
        multiplication of r1 r2 is 10/9
        division of r1  r2 is 5/2
        2/3<5/3
        5/3>=5/3
        5/3>2/3
        2/3!=5/3
        2/3<=5/3
        5/3==5/3
```

6.1.7 Overloading Unary Operators

Unary and Binary operators can be overloaded. We have seen the overloading of binary operators in the above example of overloading. Thus unary – (negation) operator, prefix ++,- - can be overloaded. There is a different method to overload the postfix ++ and --.

```
#include<iostream.h>
class space
{
int x;
int y;
int z;
public:
space(int a1,int b1, int c1)
{x = a1; y = b1; z = c1;}
```

```
void operator -()
{
x = -x;
y= -y;
z =-z;
} void operator --()
{
--x;
--y;
--z;
}  void operator -=(space& s)
{
x-= s.x;
y-= s.y;
z-= s.z;
}
void putdata()
{
cout<<"\nx:"<<x<<"\ty:"<<y<<"\tz:"<<z;
}  };
int main()
{
space s1(2,-1,3),s2(1,1,1);
cout<<"\nData in s1:";s1.putdata();
cout<<"\nData in s2:";s2.putdata();
-s1;
cout<< "\nNegation of s1:";s1.putdata();
--s1;
cout<<"\nPredecrement  s1:";s1.putdata();
s1-=s2;
cout<<"\ns1-=s2";s1.putdata();
return 0;
}
```

output:

```
Data in s1:
x:2     y:-1    z:3
Data in s2:
x:1     y:1     z:1
Negation of s1:
x:-2    y:1     z:-3
Predecrement  s1:
x:-3    y:0     z:-4
s1-=s2
x:-4    y:-1    z:-5
```

6.1.8 Overloading Of Insertion And Extraction Operators

The insertion, << and extraction >> operators can be overloaded as any other operators. The value returned must be stream. The type of value returned must have the & symbol added to the end of the type name. The prototype and function definition is as follows:

Prototypes:

```
class  classname
{
public:
        . . . . .
         . . . . .
friend istream& operator >>(istream& parameter1,
                                    classname&  parameter2);
friend ostream&operator<<(ostream& parameter3,
                          classname & parameter4);
. . . . .
private:
. . . . . .
 . . . . . .
}

#include<iostream.h>
class space
{
int x;
int y;
int z;
public:
space(int a1,int b1, int c1)
{x = a1; y = b1; z = c1;}
void operator -()
{
x = -x;
y= -y;
z =-z;
}
void operator --()
{
--x;
--y;
```

```
--z;
}
void operator -=(space& s)
{
x-= s.x;
y-= s.y;
z-= s.z;
}
friend istream& operator >>(istream &din, space&);
friend ostream& operator <<(ostream &dout,space&);
};
istream& operator >>(istream &din, space& s)
{
din>>s.x>>s.y>>s.z;
return din;
}
ostream& operator <<(ostream &dout, space& s)
{
dout<<"\nx:"<<s.x<<"\ty:"<<s.y<<"\tz:"<<s.z;
return dout;
}
int main()
{
space s1(2,-1,3),s2(1,1,1);
cout<<"\nData in s1:"<<s1;
cout<<"\nData in s2:"<<s2;
-s1;
cout<< "\nNegation of s1:"<<s1;
--s1;
cout<<"\nPredecrement of s1:"<<s1;
s1-=s2;
cout<<"\ns1-=s2"<<s1;
return 0;
}
```

output:

```
Data in s1:
x:2    y:-1    z:3
Data in s2:
x:1    y:1    z:1
Negation of s1:
x:-2    y:1    z:-3
Predecrement of s1:
x:-3    y:0    z:-4
s1-=s2
x:-4    y:-1    z:-5
```

6.2 Abstract Data Types

A **Data type** consists of a collection of values together with a set of basic operations defined on these values . If the programmer who uses the data type does not have access to the details of how the values and operations are implemented, such data type is called an abstract data type.

To define a class as an abstract data type, the specification details of the class type used by the programmer is separated from the details of implementation. In order to do this,

- All the member variables of the class are made private.
- All basic operations used by the programmer are made public member functions and their usage is completely specified.
- Any helping functions are made private.

The **interface** of an ADT tells how to use the ADT in the program. It consists of public member functions of the class along with comments how to use them. The interface is to be known to use an abstract data type.

The **implementation** of the ADT tells how this interface is realized as C++ code. It consists of private members of the class and the definitions of both public and private member functions.

6.2.1 Separate Compilation

An ADT is defined as a class and the definition and implementation of its member functions are put in separate files.

- The definition of the class is put in a header file called the interface file. The name of the file ends with .h. The interface file also contains the prototypes for the functions and overloaded operators that define basic ADT operations.

- The definitions of all functions and overloaded operators mentioned in the interface file are placed in another file called implementation file. This file must include the directive that names the interface file. The interface and implementation files have the same filename, but end with different suffixes.

- The main part of the program is placed in another file called an application file. This also must include the naming directive of the interface file. The application file must be compiled separately from the implementation file.

The object code produced by compiling the application file and object code produced by compiling the implementation file must be linked and executed.

Program using separate compilation

```
/*This program uses operator overloading of =,--(pre decrement and post decrement), < , <<,>> operators*/
//dist.h(header file) interface file
#ifndef DIST_H
#define DIST_H
enum bool{false,true};
class distance
{
        int feet;
        float inch;
        public:
        distance();
        distance( int f, float i);
        distance operator + (distance &d);
        void operator --();
        void operator --(int i);
        friend bool operator <(distance d1,distance d2);
        friend istream &operator>>(istream &ip, distance &d);
        friend ostream &operator<<(ostream &op,distance &d);
};

#endif
//dist.cpp implementation file
#include<iostream.h>
#include "dist.h"
distance::distance()
{
        feet = 0;
        inch =0;
```

```
}
distance::distance( int f, float i)
{
        feet = f;
        inch = i;
}
distance distance::operator + (distance &d)
{
        distance temp;
        temp .inch = inch + d.inch;
        temp.feet = feet + d.feet;
        while (temp.inch >= 12.0)
        {
        temp.inch -=12.0;
        temp.feet +=1;
        }
        return temp;
}
void distance::operator --()
{
        --feet;
}
void distance::operator --(int )
{
        feet--;
}

bool operator <(distance d1,distance d2)
{
        if (d1.feet < d2.feet)
        return true;
        else
        if (d1.feet > d2.feet)
        return false;
        else
        if(d1.inch < d2.inch)
        return true;
        else
        return false;
}
istream &operator>>(istream &ip, distance &d)
{
        ip>>d.feet>>d.inch;
        return ip;
}
```

```
ostream &operator<<(ostream &op,distance &d)
{
        op<<d.feet<<"\'-"<<d.inch<<"\"";
        return op;
}

//distapp.cpp   application file

#include<iostream.h>
#include"dist.h"
#include "dist.cpp"
int main()
{
        distance d1(3,9),d2,d3;
        cout<<"\nEnter distance 2:";
        cin>>d2;
        cout<<"\nFirst distance:"<<d1;
        cout<<"\nSecond distance:"<<d2;
        d3 = d1 + d2;
        cout<<"\nSum of two distances:"<<d3<<endl;
        return 0;
}
```

Output:

```
Enter distance 2:3
4

First distance:3'-9"
Second distance:3'-4"
Sum of two distances:7'-1"
```

6.3 Summary

- We have covered different tools for defining the class as an Abstract Data Types. In this, we have studied how to write friend functions and also the operator overloading of unary binary and extraction and insertion operators .
- We have also studied about the constant parameter modifiers in functions.
- We have also reviewed an Abstract data type and how to convert a class as an Abstract data type. We have also covered the separate compilation. The definition of the class and implementation of its

functions are placed in separate files. This is compiled separately and can be used in any application.

6.4 Technical Terms

Accessory Function: Member functions that give access to the private members of the class.

Friend Function: A function although not a member of a class is able to access the private members of that class.

Overloading Operator: : A language feature that allows an operator to be given more than one definition. The types of arguments with which the operator is called, determines which definition is used.

6.5 Model Questions

1. Explain the different tools for defining Abstract data types with Examples.
2. What is a friend function? How is it different from the member function?
3. What is operator overloading? Explain with an example the overloading of a binary operator.
4. Explain the overloading of << and >> operators with an example?
5. Define an Abstract Data type class in separate files.

6.6 References

Object-oriented programming with C++,
by E. Bala Gurusamy.

Problem solving with C++
by Walter Savitch

Mastering C++
by K.R.Venugopal, RajkumarBuyya, T.RaviShankar

AUTHOR

M. NIRUPAMA BHAT, MCA., M.Phil.,
Lecturer,
Dept. Of Computer Science,
JKC College,
GUNTUR.

Lesson 7:Arrays and Strings

Objectives

After completing this lesson you will understand about:

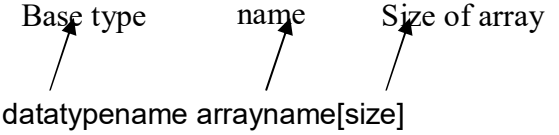
- The basics of defining, declaring, initializing and using arrays.
- Passing indexed variables and entire as arguments to functions.
- The arrays of classes and classes with arrays as member variables.
- The basics of multidimensional arrays, multidimensional array parameters and passing them into functions.
- The string basics, declaring and initializing cstring variables, predefined cstring functions and getline function.
- Defining cstring functions and array of strings.

Structure Of The Lesson

7.1 Introduction To Arrays

An array is a group of related data items of same data type that share a common name. It is used to process a collection of data, all of which is of the same type such as list of test scores, a list of temperatures, list of names etc.

Syntax:



An array declaration of the above form will define the declared size number of index variables namely arrayname[0],arrayname[1],….. arrayname[size –1]. Each index variable is a variable of type datatypename. These index variables are also known as subscripted variables or elements. The number in the square bracket is called as index number or subscript. Indexes are numbered starting with zero. The number of index variables in an array is known as the size of the array. The data type of the array is called the base type of the array.

e.g.:
```
int score[100];
double temp[50];
```

score is an integer array consisting of 100 elements starting from score[0] to score[99]. As it is an integer array, integer type of data can be stored in it.

Initializing The Arrays: An array variable can be initialized like any other variable. When initializing an array, the values of various index variables are enclosed in braces and separated with commas. All the values in the list should be of the base type of the array. This list is called an initialized list.

e.g.:
```
int score[3]={5,10,50};
```

In the above example, it is array called score with three elements where score[0]=5, score[1]=10, score[2]=50. If a fewer values than the size of the array are declared, those values will be used to initialize the first few index variables and remaining will be initialized to zero of array base type.

e.g.: int score[5]={5,10,50};
Here score[3] and score[4] are initialized to zeros.

If an array is initialized when it is declared the size of the array can be omitted. The array will be automatically declared to have the minimum size needed for the initialized variables. C++ will create the array with sufficient size to accommodate the entire initialized list.

e.g.: int score[] = { 5,10,15};

Here, the size of the array is not given. But, depending upon the number of arguments present in the array during initialization, the array is declared to be of size 3 automatically.

Write a program to read five scores and show how much each score differs from the highest score.

```
#include<iostream.h>
#include<conio.h>
int a[20];//declaration of an array
int main()
{
int i,size, max=-20;//Take max as a small value
//clrscr();
cout<<"enter size of array:";
cin>>size;
cout<< " enter "<<size<<" scores";
for(i=0;i<size;i++)
cin>>a[i];
for(i=0;i<size;i++)
if (a[i]>max)
max=a[i];
cout<<"the difference from max marks:";
for(i=0;i<size;i++)
cout<<"a["<<i<<"]="<<a[i]<<"\t"
<<"Difference:"<<max-a[i]<<"\n";
 getch();
return 0;
}
```

Output:

```
enter size of array:5
 enter 5 scores
10
20
```

```
30
40
50
the difference from max marks:
a[0]=10 Difference:40
a[1]=20 Difference:30
a[2]=30 Difference:20
a[3]=40 Difference:10
a[4]=50 Difference:0
```

7.2 Arrays In Functions

In this section we will study how array elements are sent as arguments to the functions as well as how an entire array is passed as an argument to the function.

7.2.1 Indexed Variables As Function Arguments

An indexed variable can be an argument to a function in exactly the same way that any variable can be argument.

eg: int i,n,a[10];

The variables of an array can be passed into a function as an individual data item.

Here is a sample program:

// Program to pass array element as an argument to the function

```
#include<iostream.h>
int diff(int,int);
int main()
{
int a[5],i;
int max=-20;
for(i=0; i<5;i++)
{
cout<<"enter score of each student in one subject:";
cin>>a[i];
}
for(i=0; i<5;i++)
```

```
if (a[i]>max)
max=a[i];
cout<<"The maximum score is:"<<max<<"\n";
cout<<" The scores and their difference from the maximum score:\n";
int d;
for(i=0;i<5;i++)
{
d=diff(a[i],max);
cout<<a[i]<<"\t"<<d<<"\n";
}
return 0;
}
int diff (int x, int m)
{
return(m-x);
}
```

output:

```
enter score of each student in one subject:10
enter score of each student in one subject:20
enter score of each student in one subject:30
enter score of each student in one subject:60
enter score of each student in one subject:50
The maximum score is:60
 The scores and their difference from the maximum score:
    10     50
    20     40
    30     30
    60     10
    50     10
```

Write a program to add five to each employees allowed no.of vacations.

```
#include<iostream.h>
int adjustdays(int old_days);
void main()
{
int i,vacation[5],emp;
cout<<"enter no of employees:\n";
cin>>emp;
cout<<"Enter leave details for "<<emp<<" employees";
```

```
for(i=0;i<emp;i++)
{
cin>>vacation[i];
vacation[i]=adjustdays(vacation[i]);
}
cout<<"vacations after updation:\n";
for(i=0;i<emp;i++)
{
cout<<"employee:"<<i+1<<"vacation details "<<
vacation[i]<<"days \n";
}
}
int adjustdays(int olddays)
{
return(olddays+5);
}
```

output:

```
enter no of employees:
3
Enter leave details for 3 employees
2
4
1
vacations after updation:
employee:1 vacation details 7days
employee:2 vacation details 9days
employee:3 vacation details 6days
```

7.2.2 Passing Entire Arrays As Function Arguments

An argument to a function may be an entire array, but an argument for the entire array is neither a call by value nor a call by reference argument. It is a new kind of argument known as array argument. When an array argument is plugged in for an array parameter, all that is passed to the function is the address in memory of first index variable of the array argument (i.e., one indexed by 0).

The array argument does not tell the function the size of the array. Therefore when an array parameter is passed to a function, normally another formal parameter of type int that gives the size of the array should be present.

An array argument is like a call by reference argument. If the function body changes the array parameter, when the function is called, these changes are actually made to the array arguments. Thus, the function can change the values of an array argument.

Syntax of function prototype passing an array parameter:

Returntype functionname(basetype arrayname[ ]…);

e.g.: void sumarray (double & sum,double a[],int size);

Write a program to read an array and find the sum of elements in an array.

```
#include<iostream.h>
void read(int a[], int &size);
int sum(int a[], int&size);
void display (int ,int );
void main()
{
int a[20],num,s;
read(a,num);
s=sum(a,num);
display(s,num);
}
void read (int a[],int &size)
{
cout<<"enter size of an array:";
cin >> size;
cout<<"Enter the elements:";
for(int i=0;i<size;i++)
cin>>a[i];
return;
}
int sum (int a[],int &size)
{
int s=0;
for (int i=0;i<size;i++)
s+=a[i];
return s;
}
void display (int s, int size)
{
cout<<"the sum of "<<size<<" elements is"<<s;
}
```

output:

```
enter size of an array:5
2
3
4
5
6
the sum of 5elements is20enter size of an array:5
Enter the elements:1
2
3
4
5
the sum of 5 elements is15
```

Write a program to search a number using linear search technique.

```
#include<iostream.h>
enum bool{false,true};
int search(int const arr[],int,int);
void read (int a[],int size);
int main()
{
int num,arr[30],result,target;
cout<<"enter size:";
cin >> num;
cout<<”enter elements:”;
read(arr,num);
cout<<"enter element to search";
cin >>target;
result=search(arr,num,target);
if(result==-1)
cout << target<<"not found";
else
cout << target <<" found at"<<(result+1);
return 0;
}
void read (int a[],int num)
{
for(int i=0;i<num;i++)
cin>>a[i];
}
int search(int const arr[],int num,int tar)
{
```

```
bool found=false; int i=0;
while((!found)&&(i<num))
{
        if (tar==arr[i])
found=true;
else
i++;
}
if(found==true)
return i;
else
return -1;
}
```

output:

```
enter size:4
enter elements:
2
6
4
1
enter element to search4
4 found at3
```

Program to sort the given integers using selection sorting technique.

```
#include<iostream.h>
void read (int a[],int &size);
void display (int a[],int &size);
void swap(int& v1,int& v2);
void main()
{
int arr[20],numb;
int minindex,min;
read(arr,numb);
cout<<" Array before sorting:\n";
display(arr,numb);
int i;
for(i=0;i<numb-1;i++)
{
min=arr[i];
minindex=i;
```

```
for (int j=i+1;j<numb;j++)
if(arr[j]<min)
{
min=arr[i];
minindex=j;
}
swap(arr[i],arr[minindex]);
}
cout<<"\nArray after sorting:\n";
display (arr,numb);
}
void read (int a[],int &size)
{
cout<<"enter size of an array:";
cin >> size;
for(int i=0;i<size;i++)
cin>>a[i];
return;
}
void swap(int& v1,int& v2)
{
int temp;
temp = v1;
v1 = v2;
v2 = temp;
}
void display (int a[],int &size)
{
for(int i=0;i<size;i++)
cout<<a[i]<<"\n";
return;
}
```

output:

```
enter size of an array:4
5
3
6
2
 Array before sorting:
5
3
6
2
Array after sorting:
```

2
3
5
6

7.3 Arrays And Classes

We can also declare the arrays of classes and classes with arrays as member variables.

Arrays of classes: The base type of an array can be a class. We can declare an array of variable of that class type then each array element will be an object of the class. The syntax for declaring an array of object is

Syntax: **Classname arrayname[size];**
e.g: student s1[10];

Program to illustrate the arrays of objects

```
#include <iostream.h>
class employee
        {
        char name[30];
        float age;
    public:
        void getdata(void);
        void putdata(void);
        };
void employee::getdata(void)
{
cout<<"Enter name";
cin>>name;
cout<<"Enter age";
cin>>age;
}
void employee::putdata(void)
{
cout<<"Name: "<<name<<"\n";
cout<<"Age: "<<age<<"\n";
}
const int size = 2;
```

```
int main( )
{

   employee manager[size];//Array of managers
   int i;

   for (i = 0; i < size; i++)
   {
         cout<<"\nDetails of manager"<<i+1<<"\n";
         manager[i].getdata();
   }
   cout<<"\n";
   for (i = 0; i < size; i++)
      {
         cout<<"\nManager"<<i+1<<"\n";
         manager[i].putdata();
         }
   return 0;
}
```

Output:

```
Details of Manager1
Enter name:harish
Enter age:40

Details of Manager1
Enter name:girish
Enter age:42

Manager1
Name :harish
Age:40

Manager2
Name :girish
Age:42
```

Array as class members: Arrays can be declared as members of a class.They can be of any data type. The arrays within the class can be private, public or protected. Then every object of that class will contain a copy of array declared in the class.

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

```
char name[20];
int rno;
int marks[3];
public:
void read()
{
cout<<"Enter name:";
cin>>name;
cout<<"Enter rno:";
cin>>rno;
cout<<"Enter 3 marks:";
for(int i = 0;i<3;i++)
{
  cin>>marks[i];
}
}
void display()
{
cout<<"Name: "<<name<<"\n";
cout<<"RNo: "<<rno<<"\n";
for(int i = 0;i<3;i++)
cout<<marks[i]<<"\t";
cout<<"\n";
}
};

void main()
{
student stu;
stu.read();
stu.display();
}
```

Output:

```
Enter name:rama
Enter rno:1
Enter 3 marks:90
80
70
Name: rama
RNo: 1
90      80      70
```

7.4 Multidimensional Arrays

It is sometimes useful to have more than one index, and this is allowed in C++ with multidimensional arrays. A multidimensional array is an array of arrays. An array with more than one subscript or size specifier is defined as multidimensional array.

Syntax: Base type dataname[size 1][size 2]….[size n];
e.g.: int mat[2][3];

In the above example, mat is a two dimensional matrix with 2 rows and 3 columns. The index variables of the array mat are:
mat[0][0] , mat[0][1], mat[0][2]
mat[1][0] , mat[1][1], mat[1][2]
The number of elements in a multi dimensional array is equal to the product of all its subscripts. Thus the above matrix has 2X 3 elements.
The indexes start from 0 to size-1.

7.4.1 Accessing Multidimensional Array Elements

Two indices or subscripts are used to access two-dimensional arrays, three for three-dimensional arrays and so on. The values of the elements of the two dimensional arrays are accessed by specifying the name of the variable, row and column index of the element.

7.4.2 Initialization Of Multidimensional Arrays

Consider the array marks[5][4]. It is an integer array. The first subscript indicates rollno of a student, second subscript indicates marks obtained by each student in 4 subjects. Thus this array has 5 rows, 4 columns and with subscripts 0 to 4 & 0 to 3. This array can be initialized as

```
int marks[5][4]={{20,40,60,70}, {3,50,40,60} ,{4,9,10,16},
                 {10,,60,20,30} ,  {50,80,60,90}};
```

7.4.3 Storage of two dimensional arrays

The two dimensional arrays are stored in two ways. They are row major order, column major order.
For e.g. **:** let us consider a 3x4 integer matrix, mat. The different indexed variables of the matrix is as follows:

mat[0][0]	mat[0][1]	mat[0][2]	mat[0][3]
mat[1][0]	mat[1][1]	mat[1][2]	mat[1][3]
mat[2][0]	mat[2][1]	mat[2][2]	mat[2][3]

Let us see how this matrix is stored in the computers memory. The computer's memory consists of list of numbered location called bytes. The variable is represented as a portion of this memory. Thus, a variable is described by two pieces of information: an address in memory (giving the location of the first byte for that variable) and the type of the variable. The location of various array-indexed variables are always placed next to one another in memory. When the array is declared, the computer reserves enough memory to hold the variables of the array, depending on the data type. The computer remembers the address of the first variable mat[0][0].

Let us consider the row major order. There are four columns the starting address of k th row will be

Base + k th row x no. of columns x size of the data type

Let the first element of the array is stored at the address location 200. For an element a[2][0] it is stored that

200+2*4*2
200+16=216

To get the full address of an array variable we use

base+(rowindex X total no.of columns) X size of the datatype) + columnindex* size of data type.

Similarly in column major order the storage of a[I][j] is given as

Base + (columnindex X total no of rows) * size of datatype)+ rowindex X size of datatype

7.4.4 Passing Multidimensional Arrays Into Functions

When a multidimensional array parameter is given in a function heading or prototype, the size of the first dimension is not given, but the remaining dimension sizes must be given in square brackets. Since the first dimension size is not given, an additional parameter of type int, which gives the size of the first dimension is needed. Here is an example of a function prototype with a two dimensional array parameter:

```
//Program to read and display the quiz scores of given students
void readarray(int  p[][100], int sizedimension);


#include <iostream.h>
#include <iomanip.h>
const int NUMBER_STUDENTS = 4, NUMBER_QUIZZES = 3;
void compute_st_ave(const int grade[][NUMBER_QUIZZES], double
st_ave[]);
void compute_quiz_ave(const int grade[][NUMBER_QUIZZES], double
quiz_ave[]);
void display(const int grade[][NUMBER_QUIZZES],
                    const double st_ave[], const double quiz_ave[]);

int main( )
{
    int grade[NUMBER_STUDENTS][NUMBER_QUIZZES];
   double st_ave[NUMBER_STUDENTS];
   double quiz_ave[NUMBER_QUIZZES];

   grade[0][0] = 10; grade[0][1] = 10; grade[0][2] = 10;
   grade[1][0] = 2;  grade[1][1] = 0;  grade[1][2] = 1;
   grade[2][0] = 8;  grade[2][1] = 6;  grade[2][2] = 9;
   grade[3][0] = 8;  grade[3][1] = 4;  grade[3][2] = 10;

   compute_st_ave(grade, st_ave);
   compute_quiz_ave(grade, quiz_ave);
   display(grade, st_ave, quiz_ave);
   return 0;
}

void compute_st_ave(const int grade[][NUMBER_QUIZZES], double
st_ave[])
```

```
{
   for (int st_num = 1; st_num <= NUMBER_STUDENTS; st_num++)
   {//Process one st_num:
      double sum = 0;
 for (int quiz_num = 1; quiz_num <= NUMBER_QUIZZES; quiz_num++)
         sum = sum + grade[st_num-1][quiz_num-1];
 //sum contains the sum of the quiz scores for student numberst_num.
      st_ave[st_num-1] = sum/NUMBER_QUIZZES;
      //Average for student st_num is the value of st_ave[st_num-1]
   }
}
void compute_quiz_ave(const int grade[][NUMBER_QUIZZES], double
quiz_ave[])
{
for (int quiz_num = 1; quiz_num <= NUMBER_QUIZZES; quiz_num++)
   {//Process one quiz (for all students):
      double sum = 0;
for (int st_num = 1; st_num <= NUMBER_STUDENTS; st_num++)
         sum = sum + grade[st_num-1][quiz_num-1];
//sum contains the sum of all student scores on quiz number quiz _
num.
      quiz_ave[quiz_num-1] = sum/NUMBER_STUDENTS;
    //Average for quiz quiz_num is the value of quiz_ave[quiz_num-1]
   }
}

//Uses iostream and iomanip:
void display(const int grade[][NUMBER_QUIZZES],
                    const double st_ave[], const double quiz_ave[])
{
    cout.setf(ios::fixed);
   cout.setf(ios::showpoint);
   cout.precision(1);

   cout << setw(10) << "Student"
        << setw(5) << "Ave"
        << setw(15) << "Quizzes\n";
   for (int st_num = 1; st_num <= NUMBER_STUDENTS; st_num++)
   {//Display for one st_num:
      cout << setw(10) << st_num
           << setw(5) << st_ave[st_num-1] << " ";
      for (int quiz_num = 1; quiz_num <= NUMBER_QUIZZES;
quiz_num++)
         cout << setw(5) << grade[st_num-1][quiz_num-1];
      cout << endl;
```

```
   }
   cout << "Quiz averages = ";
   for (int quiz_num = 1; quiz_num <= NUMBER_QUIZZES;
quiz_num++)
      cout << setw(5) << quiz_ave[quiz_num-1];
      cout << endl;
}
```

Output:

```
  Student  Ave      Quizzes
       1 10.0    10   10   10
       2  1.0     2    0    1
       3  7.7     8    6    9
       4  7.3     8    4   10
Quiz averages =   7.0  5.0  7.5
```

7.5 Strings

A string is an array of characters terminated by '\0' (null character). There is already a predefined string class in the Standard Library, whose members are declared in the <string> header. To distinguish the strings from the string class, the earlier strings are referred as cstrings. However, the cstring places a special character '\0' immediately after the last character of the string. The declaration of cstring variable is similar to the character array variable.

Syntax: char cstringname[maxsize+1];
e.g.: char name [15];

The "+1" allows space for null character, which terminates any cstring stored in array. The cstring variables can be initialized when it is declared.

e.g.: char str[ ]="hello";
char str1[20]="hello";

The above statement initializes the cstring variables. The first statement declares a cstring "str" as a character array of size 6. In the second initialization str, is an array of size 20 and places or stores the value "hello" from 0 to 5^{th} index variable including null character ('\0').

7.5.1 Index Variable Of cstrings

As a cstring is an array, it has index variables. These variables are used like those of any other array.

Ex: - str[ ]=”hello”;

This is same as

```
str[0]=’h’;
str[1]=’e’;
str[2]=’l’;
str[3]=’l’;
str[4]=’o’;
str[5]=’\0’;
```

The index variable of the cstring can be manipulated in all possible ways but the null character should be present at the end of the stream.

Write a program to read a string of characters and display it by adding ‘2’ to each character.

```
#include<iostream.h>
void main ( )
 {
    char s;
     int n;
     cout<<"enter how many characters:";
     cin>>n;
     int i=1;
     while(i<=n)
     {
       cout<<"enter character:";
       cin>>s;
       if(s=='y'||s=='z')
        {
            cout<<char (s-24)<<endl;
          }
       else
           cout<<char (s+2)<<endl;
            i++;
      }

   }
```

output:

```
enter how many characters:3
```

```
enter character:e
g
enter character:t
v
enter character:z
b
```

7.5.2 Operation On Strings

Reading a cstring: The value of a cstring variable can read using "cin" with the extraction operator. The extraction operator reads the input stream into the cstream variable until the first blank in the input is accessed.

e.g.: -
```
char name [20];
cin>>name;
```
In the above statement if we give "JKC college" as input it reads only JKC, as a blank is encountered.

Reading input including blank space: A member function getline can be used to read a line of input with blank spaces and place the string of characters into a string variable.

Syntax: Inputstreamobject. getline (cstring var,size,delimiting sy)

e.g.:- cin.getline (name, 20,'*');

The string terminates when '*' is encountered.

String output: A string output statement sends the character until it finds the null character. It will not print the character after the null character if there are such characters. If the output statement does not find the null character, it continues to insert the characters into the output stream giving unpredictable output.

e.g.:
```
cout<<name;
```

7.5.3 cstring Predefined Functions

The predefined cstring functions are found in <string.h> or <cstring> library file.

Length of a string: The length of a string can be found using strlen().

Syntax: strlen(string);

It returns an integer equal to the length of the source string. It dose not count the null character.

e.g.: y=strlen("hello");

Copying a string: One cstring variable can be copied into another cstring variable using strcpy() function.

Syntax:

Strcpy (target string,source string);

```
e.g.:  char x[ ]="hello";
       char y[ ];
        Strcpy(y,x);
```

It copies the source string values into the target string. This function does not check the size of target string variable, if it is large enough to hold the value of source string.

Note: A cstring variable cannot be used with assignment operators.

Comparing cstrings:

Two strings can be compared using strcmp() function .

Syntax: strcmp(cstring1,cstring2);

This function returns 0, if string1 and string2 are same. It returns a value less than 0 if string1 is less than string2 and greater than 1, if string1 is greater than string2. The strings are compared in lexicographic ordering, where (a<b<……..<z).

```
eg:    char name1[10],name2[10];
   cin.getline(name1,10);cin.getline(name2,10);
    X=strcmp(name1,name2);
```

Concatenating two strings: Two strings can be concatenated using strcat function.

Syntax:

Stract(targetstring,src_string);

It is to form by a longer cstring by placing the two shorter cs trings end-to-end. The first argument must be a cstring variable. The second argument can be anything that evaluates to a cstring value, such as quoted string. The result is placed in the cstring variable that is the first argument.

e.g.:

```
char stringvar[20] =”The rain ”;
 strcat(stringvar,”in spain”);
```

This code will change the value of stringvar to “The rain in Spain” does not check to see that target stringvar is large enough to hold the result of the concatenation.

7.5.4 Defining cstring Functions

A cstring variable is an array, so a cstring parameter to a function is simply an array parameter. The size of the cstring variable should be included, whenever a function changes the value of a string parameter. The null character is used to detect the end of the string value that is stored in the cstring variable.

7.5.5 Array Of Strings

In C++ an array of cstrings is represented as a two dimensional array of characters. For example, the following declares an array called name, which can hold a list of five names, with at most 19 characters with one indexed variable holding ‘\0’ (null character).

```
char  name [5][20];
```

An array of cstrings can be manipulated by using both indexes simultaneously, but it is nice to manipulate only one index at a time. A list of cstrings can be manipulated by a loop that steps through values of first index and treats each indexed variable – such as name[0], name[1], name[2] and so forth- as a single cstring variable that can be manipulated by some cstring function.

//Program to pass cstrings into functions

```
#include<iostream.h>
#include<string.h>
void read(char x[][10],int y);
void write(char x[][10],int y);
void main()
{
char s[5][10];
read(s,5);
write(s,5);
}
void read(char x[][10],int y)
{
for(int i = 0;i<y;i++)
cin.getline(x[i],10);
        }
void write(char x[][10],int y)
{
for(int i = 0;i<y;i++)
cout<<x[i]<<"\n";
}
```

output:

```
hello
hai
bye bye
see you
good day
hello
ha
ibye bye
see you
good day
```

//Program to sort a given array of strings

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

void read(char x[][10],int y)
{
for(int i = 0;i<y;i++)
cin.getline(x[i],10);
}
void write(char x[][10],int y)
```

```
{
for(int i = 0;i<y;i++)
cout<<x[i]<<"\n";
}
void sort(char x[][10],int y)
{
int i,j;
for(i = 0;i<y-1;i++)
{
for(j=0;j<(y-1-i);j++)
{
if (strcmp(x[j],x[j+1]) >0)
{
char t[10];
strcpy(t,x[j]);
strcpy(x[j],x[j+1]);
strcpy(x[j+1],t);
}
}
}
}
void main()
{
int x=3;
char arr[5][10];
cout<<"Enter the no of strings;";
read(arr,x);
sort(arr,x);
cout<<”Strings after sorting;\n”;
write(arr,x);
}
```

output:

```
Enter the 3 strings:rama
krishna
govinda

 Strings after sorting:
govinda
krishna
rama
```

//Program to overload string operations

```
#include<iostream.h>
#include<string.h>
class string
{
char str[40];
public:
string()
{
strcpy(str,'\0');
}
string(char x[])
{
strcpy(str,x);
}
          void operator=(string s)
          {
strcpy(str,s.str);
}
int operator ==(string s)
{
 if((strcmp(str,s.str))!= 0)
 return 0;
 else
 return 1;
}
int operator <(string s)
{
if((strcmp(str,s.str))< 0)
 return 1;
 else
 return 0;
 }
friend ostream& operator<<(ostream& out,string s);
friend istream& operator>>(istream& in, string &s);
friend string operator+(string ss1,string ss2);
};
string operator+(string ss1,string ss2)
{
strcat(ss1.str,ss2.str);
return ss1;
}
ostream& operator<<(ostream& out,string s)
{
```

```
out<<s.str;
return out;
}
istream& operator>>(istream& in,string &s)
{
in>>s.str;
return in;
}

void main()
{
string s1,s2("Hello"),s3,s4;
cout<<"Enter two strings:";
cin>>s3>>s4;
s1 = s2+s3+s4;
cout<<"s1 after concatenating s2,s3,s4:"<<s1<<endl;
cout<<"s2"<<s2<<endl;
if (s3 == s4)
cout<<"s3 and s4 r equal\n";
else
cout<<"s3 and s4 are different\n";
if(s3<s4)
cout<<s3<<" is less than "<<s4<<endl;
}
```

output:

s1 after concatinating s2,s3 and s4:Helloramakrishna

s2:Hello

s3:rama

s4:krishna

s3 and s4 are different

7.6 C++ Standard String Class

C++ provides a simple, safe alternative to using chars to handle strings. The C++ string class, part of the std namespace, allows to manipulate Strings safely.

Declaring a string is easy:

```
using namespace std;
string my_string;
```

or
std::string my_string;

An initial value for the string can be specified in a constructor:
using namespace std;
string my_string("starting value");

String I/O is easy, as strings are supported by cin.
cin>>my_string;

To read an entire line at a time, the getline function can be used and passed in an input stream object (such as cin, to read from standard input, or a stream associated with a file, to read from a file), the string, and a character on which to terminate input. The following code reads a line from standard input (e.g., the keyboard) until a new line is entered.

using namespace std;
getline(cin, my_string, '\n');

Strings can also be assigned to each other or appended together using the + operator:

string my_string1 = "a string";
string my_string2 = " is this";
string my_string3 = my_string1 + my_string2;

// Will ouput "a string is this"
cout>>my_string3>>endl;
Naturally, the += operator is also defined

String Comparisons: One of the most confusing parts of using char*s as strings is that comparisons are tricky, requiring a special comparison function, and using tests such as == or < don't mean what you'd expect. Fortunately, for C++ strings, all of the typical relational operators work as expected to compare either C++ strings or a C++ string and either a C string or a static string (i.e., "one in quotes").

For instance, the following code does exactly what you would expect, namely, it determines whether an input string is equal to a fixed string:

```
string passwd;
getline(cin, passwd, '\n');
if(passwd == "xyzzy")
{
   cout<<"Access allowed"; }
```

String Length and Accessing Individual Elements: To take the length of a string, you can use either the length or size function, which are members of the string class, and which return the number of characters in a string:

```
string my_string1 = "ten chars.";
int len = my_string1.length(); // or .size();
```

Strings, like cstrings can be indexed numerically.

For instance, you could iterate over all of the characters in a string indexing them by number, as though the string were an array.

Note that the use of the length() or size() function is important here because C++ strings are not guaranteed to be null-terminated (by a '\0'). (In fact, you should be able to store bytes with a value of 0 inside of a C++ string with no adverse effects. In a cstring, this would terminate the string!)

```
int i;
for(i = 0; i < my_string.length(); i++)
{
   cout<<my_string[i];
}
```

Incidentally, C++ string iterators are easily invalidated by operations that change the string, so be wary of using them after calling any string function that may modify the string.

Searching and Sub strings: The string class supports simple searching and sub string retrieval using the functions find(), rfind(), and substr(). The find member function takes a string and a position and begins searching the string from the given position for the first occurrence of the given string. It returns the position of the first occurrence of the string, or a special value, string :: npos, that indicates that it did not find the sub string.

```
int find(string pattern, int position);
```

This sample code searches for every instance of the string "cat" in a given string and counts the total number of instances:

```
string input;
int i = 0;
int cat_appearances = 0;
```

```
getline(cin, input, '\n');

for(i = input.find("cat", 0); i != string::npos; i = input.find("cat",
i))
{
   cat_appearances++;
   i++;  // Move past the last discovered instance to avoid
finding same
        // string
}
cout<<cat_appearances;
```

Similarly, it would be possible to use rfind in almost the exact same way, except that searching would begin at the very end of the string, rather than the beginning.

On the other hand, the substr function can be used to create a new string consisting only of the slice of the string beginning at a given position and of a particular length:

```
// sample prototype
string substr(int position, int length);
```

For instance, to extract the first ten characters of a string, you might use:

```
string my_string = "abcdefghijklmnop";
string first_ten_of_alphabet = my_string.substr(0, 10);
cout<<The first ten letters of the alphabet are "
  <<first_ten_of_alphabet;
```

Typical calls to members of the Standard Class string:

Members	**Remarks**
Constructors	
string str;	Default constructor creates empty string objects.
string str(“string”);	Creates a string object with data “string”.
string str(“aString”);	Creates a string object str that is a copy of a string, which is an object of the class string.

Elements access:

str[l]	Read/write access to character at index i.
str.substr(position, length)	Returns sub string of calling object starting at position for length characters (read-only access).
str.c_str()	Returns read-only access to the cstring of data in string str.
str.at(i)	Returns read/write reference to character in str at index i.

Assignment/modifiers:

str1-str2;	Allocates space and initializes it to str2's data, release memory allocated for str1,sets str1's size to str2.
str1 +=str2	Character data of str2 is concatenated to the end of str1;the size is set appropriately.
str.empty()	Returns true if str is an empty string, false if it is not empty.
str1+str2	Returns a string that has str2's data concatenated onto the end of str1's data.The size is set appropriately.
str.insert(pos, str2)	Inserts str2 into str beginning at position pos.
str.remove(pos, len)	Removes sub string of len, starting at position pos.

Comparisons:

str1==str2 str1!=str2	Compare for equality or inequality ;returns a Boolean value.
str1<str2 str1>str2 str1<=str2 str1>=str2	All are lexicographical comparisons

str.find(str1)	Returns index of the first occurrence of str1 in str.
str.find(str1,pos)	Returns index of the first occurrence of string str1 in str; the search starts at position pos.

str.find_first_of(str1,pos) Finds first instance of any character in str1 in str, starting the search at position.

Str.find_first_not_of(str1,pos) Finds first instance of any character not in str1, in str, starting search at position pos.

7.7 Summary

- We have studied the definition of arrays, initializing and passing arrays into functions.
- We have also covered the details about the array of classes and arrays as members of class.
- The initialization and accessing of multidimensional arrays, passing the multidimensional arrays into functions are covered in detail.
- The details regarding the cstrings, operations on cstrings, predefined cstring functions, arrays of strings are covered.
- C++ Standard string class and typical calls to members of the standard class string are covered.

7.8 Technical Terms

Array: A collection of data elements arranged to be indexed in one or more dimensions. They are stored in contiguous memory.

Cstring: Array of characters that end with '\0'.

Multidimensional Arrays: Arrays with 2 or more dimensions .

7.9 Model Questions

1. What are arrays? Explain them in detail.
2. What are array of objects? How are they defined in C++?
3. What are multidimensional arrays? Explain them.

4. What is a cstring? How is it different from character array?
5. Explain some predefined string functions in C++.
6. Explain the passing of arrays into functions.
7. Explain the Standard Class string.

7.10 References

Object-oriented programming with C++,
by E. Bala Gurusamy.

Problem solving with C++
by Walter Savitch

Mastering C++
by K.R.Venugopal, RajkumarBuyya, T.RaviShankar

AUTHOR

M. NIRUPAMA BHAT, MCA., M.Phil.,
Lecturer,
Dept. Of Computer Science,
JKC College,
GUNTUR.

Lesson: 8 : Pointers and Dynamic

Objectives:

To:

- Learn about Pointers in C++
- Become aware of the basic properties of pointers
- Explore the application of pointers
- Learn about Dynamic Arrays
- Discover how to create and manipulate a dynamic array
- Learn how to use a dynamic array

Structure of the Lesson:

8.1. Introduction to pointers and pointer

Pointers are variables that hold addresses in C and C++. They provide much power and utility for the programmer to access and manipulate data in ways not seen in some other languages. They are also useful for passing parameters into functions in a manner that allows a function to modify and return values to the calling routine. When used incorrectly, they also are a frequent source of both program bugs.

We can define a variable in C++ to store a *memory address*. A **pointer** in **C++** is said to "point to" the memory address that is stored in it. Also, when defining a ***C++*** pointer variable, we must specify the type of variable to which it is pointing. For example, to define a pointer, which will store a memory address at which exists an int, we can do the following:

```
//Sample program for c++ pointer
 main()
{
    int* p;
    // p contains no particular value in this C++
code.
}
```

The asterisk in the above specifies that we have a pointer variable. Let's say we want to define an int variable and then we want to define a pointer variable, which will store the memory address of this int:

```
//c++ pointer using an int variable

main()
{
   int number(7);
   int* p_number;
   // p_number contains no particular value.
   p_number = &number;
   //Now, p_number in this c++ program
contains the

   //memory address of the variable myval
}
```

With **&number**, & is referred to as "the address-of operator". The expression &number is of the c++ type int*. We then store this int* number in our int* variable, which is p_number. Now, we will actually use this pointer:

```
//Sample program for c++ pointer

signed main()
{
   int number = 7;
   int* p_number = &number;
   *p_number = 6;
}
```

With *p_number = 6, the asterisk is referred to as "the dereference operator". It turns the expression from an int* into an int. The statement has the effect of setting the value of number to 6.

Pointers to Pointers: An int* c++ pointer points to an int, so an int** points to an int*. The variable p_p_n below stores a memory address. At that memory address exists a

variable of type int*. This int* variable also stores a memory address, at which exists an int.

//sample for c++ pointer to pointers .

```
int main()
{
   int n(7);
   int* p_n = &n;
   int** p_p_n(&p_n);
   int*** p_p_p_n = &p_p_n;
}
```

8.2. Dynamic Variables

In C++, space in memory for variables may be either statically or dynamically allocated. Statically allocated objects are those that are not created with the memory allocator **new**, that is, they are just the ordinary objects we use.

```
int x;
float money;
Employee emp;
```

These objects are of fixed, known size and the compiler arranges the required space as it turns source code into an object program. Statically allocated objects that are of local scope are put into a memory space known as the stack. Statically allocated objects of global scope live in the global address space. The key point is that for these objects their size is fixed at compile time.

Sometimes we don't know the size of an object until the program execution. Examples of this are a buffer to hold a

block of text of variable size, or an array with an undetermined number of elements, You could try to size the buffer or array to be large enough to hold the worst case, that is, to be big enough to hold anything we should encounter. But, there are two problems with this strategy. First, it consumes memory unnecessarily. Second, we can never be sure that the object is large enough, no matter how much memory is statically set aside for our object. This is a serious problem. The solution to this problem is dynamic memory allocation.

Allocating Single Objects

During program execution dynamically allocated memory comes form a pool of memory known as the heap or free store. It is allocated using the C++ operator "new" and freed using the operator "delete". To see how this works let's dynamically allocate some objects of intrinsic data types.

```
int *IDpt = new int;
float *theMoney = new float;
char *letter = new char;
```

The "new" operator returns the address to the start of the allocated block of memory. This address must be stored in a pointer. New allocates a block of space of the appropriate size in the heap for the object. For instance, "new int" reserves four bytes (on most operating systems), while "new char" reserves a single byte. Also, notice that the reserved block of memory is anonymous; it has no identifier (name). Dynamically allocated memory is accessed indirectly via a pointer. It is possible for new to fail. This will be the case if no memory is available. In this case new throw a "bad_alloc" exception.

Objects dynamically allocated using the above syntax are uninitialized. They contain whatever random bits happen to

be at their memory location. Before use, a value must be assigned.

```
int *IDpt = new int;
*IDpt = 5;
```

Alternatively, C++ provides a syntax, which initializes the allocated object via the "new" operator.

```
int *IDpt = new int(5);     //Allocates an int object and
initializes it to value 5.
char *letter = new char('J');
```

Dynamically allocated objects introduce a new twist; they must be explicitly deleted when no longer needed by a program. This is done using the "delete" operator. Delete releases the memory used by the object. That memory is then available for reuse.

```
delete IDpt;
delete theMoney;
delete letter;
```

Memory for statically allocated variables is reclaimed when they go out of scope. For instance, when execution enters a function, the function's statically allocated local variables are created on the stack. When the function exits, these variables are popped off the stack and the memory they occupied is available for reuse. The memory of dynamically allocated objects is not automatically released. It must be explicitly released using the "delete" operator. Suppose we have a function that dynamically allocated some variables

and that we neglect to call delete before exiting that function.

```
void fun()
{
   int *pt;
   int ordinaryVariable;

   pt = new int(1024);
   ....
   ....
   // dynamic variable not deleted.
}
int main()
{
   while (some condition exists) // Pseudo-code
   {       fun();
   }
   return 0;
}
```

"ordinaryVariable" is created on the stack when entering the function and popped off when exiting the function, so there is no problem. Likewise for "pt". "pt" is a local variable in the function. It holds the address of the dynamically allocated object. When the function exists, pointer is popped off the stack like any local variable, but the dynamically allocated object still exists. We just no longer have a pointer to it. Since we no longer have the pointer, the memory of the dynamically allocated object can no longer be released using delete. This is known as a memory leak. As the program continued to operate, more and more memory will be lost from the heap (free store). If the program runs long enough, eventually no memory will be available, and the program will no longer operate. Additionally, even if we don't run out of memory, the reduced pool of available memory affects system performance. The moral of all this: Be sure to delete.

Every new should be paired with a delete in your code to avoid memory leaks.

8.3. Dynamic Arrays

Dynamically Allocating Arrays
Arrays of built-in and user-defined data types may be dynamically allocated. User-defined data types include classes.

```
int *pt = new int[1024];
 //allocates an array of 1024 ints
double *dbs = new double[1000];    /* Allocates an
array of 1000 doubles to hold the amounts of the bills
*/
```

Observe the difference between:

```
int *pt = new int[1024];
 //allocates an array of 1024 ints
int *pt = new int(1024);
//allocates a single int with value 1024
```

A dynamically allocated array is best initialized using a loop, as follows.

```
int *list = new int[1024];
for (i = 0; i < 1024; i++)
{
   *list = 52; //Assigns 52 to each element;
   list++;
}
```

or equivalently

```
int *list = new int[1024];
for (i = 0; i < 1024; i++)
{
   list[i] = 52; //Assigns 52 to each element;
}
```

The syntax of the delete operator to delete dynamically allocated arrays is slightly different from what we saw for single objects.

```
delete[] pt;
delete[] dbs;
```

The square brackets after the delete tell the compiler to delete a dynamic array rather than a single object.

Dangling Pointers
Take a look at this snippet of code.

```
int *dptr, *dup;

dptr = new int(10);
dup = dptr;
cout << "The value of dptr is " << *dptr << endl;

delete dup;
*dptr = 5;
cout << "The value of dptr is " << *dptr << endl;
```

In the above example dptr is a dangling pointer. We have released the memory of the object whose address dptr holds and then continued to use it. This problem is that although the program may run, this section of memory may be used by another dynamic object allocated after the delete. The values in that object will be corrupted by the continued use of dptr. This is a very subtle programming bug and is very difficult to isolate. To avoid this bug, always set a pointer to 0, after the delete is called. Subsequent attempts to use the pointer will result in a run-time exception. This will immediately allow the bug to be identified and fixed. The corrected code is given below.

```
int *dptr;
dptr = new int(10);
cout << "The value of dptr is " << * dptr << endl;

delete dptr;
dptr = 0;
*dptr = 5;
//This statement will cause an run-time exception, now.

cout << "The value of dptr is " << * dptr << endl;
```

An example program on dynamic arrays is given below.

```
//Sorts a list of numbers entered at the keyboard.
#include <iostream.h>
#include <stdlib.h>
#include <stddef.h>

typedef int* IntArrayPtr;

void fill_array(int a[], int size);
//Precondition: size is the size of the array a.
//Postcondition: a[0] through a[size-1] have been
//filled with values read from the keyboard.

void sort(int a[], int size);
/*Precondition: size is the size of the array a.
The array elements a[0] through a[size - 1] have values.
Postcondition: The values of a[0] through a[size-1] have
been rearranged so that a[0] <= a[1] <= ... <=
a[size-1].*/

//The following prototypes are to use in the definition of
//sort:

void swap_values(int& v1, int& v2);
//Interchanges the values of v1 and v2.

int index_of_smallest(const int a[], int start_index, int
number_used);
//Precondition: 0 <= start_index < number_used.
//Referenced array elements have values.
//Returns the index i such that a[i] is the smallest of the
values
//a[start_index], a[star_index + 1], ..., a[number_used -
1].

int main( )
{
   cout << "This program sorts numbers from lowest to
```

```
highest.\n";

    int array_size;
    cout << "How many numbers will be sorted? ";
    cin >> array_size;

    IntArrayPtr a;
    a = new int[array_size];
    if (a == NULL)
    {
        cout << "Error: Insufficient memory.\n";
        exit(1);
    }

    fill_array(a, array_size);
    sort(a, array_size);

    cout << "In sorted order the numbers are:\n";
    for (int index = 0; index < array_size; index++)
        cout << a[index] << " ";
    cout << endl;

    delete [] a;

    return 0;
}




//Uses the library iostream.h:
void fill_array(int a[], int size){
    cout << "Enter " << size << " integers.\n";
    for (int index = 0; index < size; index++)
        cin >> a[index];
}

void sort(int a[], int size)
{
```

```
    int index_of_next_smallest;
         for (int index = 0; index < size - 1; index++)
    {//Place the correct value in a[index]:
        index_of_next_smallest =
                        index_of_smallest(a, index, size);
        swap_values(a[index], a[index_of_next_smallest]);
        //a[0] <= a[1] <=...<= a[index] are the smallest of
the original array
        //elements. The rest of the elements are in the
remaining positions.
    }
}

void swap_values(int& v1, int& v2)
{
    int temp;
    temp = v1;
    v1 = v2;
    v2 = temp;
}

int index_of_smallest(const int a[], int start_index, int
number_used)
{
    int min = a[start_index],
        index_of_min = start_index;
    for (int index = start_index + 1; index < number_used;
index++)
        if (a[index] < min)
        {
            min = a[index];
            index_of_min = index;
            //min is the smallest of a[start_index] through
a[index]
        }

    return index_of_min;
}
```

How to avoid Memory leaks and Dangling pointers:

A memory leak happens when you forget to free a block of memory allocated with the new operator or when you make it impossible to do so. As a consequence your application may eventually run out of memory and may even cause the system to crash. The following are a few practices to avoid memory leaks and dangling pointers

- **Delete a dynamic array before reallocating it**

```
char *string;
string = new char[20];
string = new char[30];
delete[] string;
```

In the above example, we have two consecutive memory allocations to the string pointer. This leads to memory leak. We should have a delete [] statement after the first allocation and then try to reallocate using a different size parameter. If we choose not to, the second allocation will assign a new address to the string pointer while the previous one will be lost. This makes it impossible to free the first dynamic variable further on in the code, resulting in a memory leakage. The corrected code is:

```
char *string;
string = new char[20];
delete[] string;
string = new char[30];
delete[] string;
```

- **Be sure you have a pointer to each dynamic variable**

  What do you think will happen at the end of this code fragment?

```
char *first_string = new char[20];
char *second_string = new char[20];
strcpy(first_string, "leak");
second_string = first_string;
strcpy(second_string, first_string);
delete [] second_string;
```

There is a memory leak in the above example, because you have lost the address of the dynamic variable associated with *second_string* (as a side-effect of the pointer assignment) so you cannot delete it from the heap anymore. Thus the last line of code only frees the dynamic variable associated with *first_string*, which is not what we wanted.

The main idea is to try and not lose the addresses of dynamic variables, so that you will be able to free them, after their purpose is over.

- **look for local pointers**

  Consider the following function :

```
Void leak() {
   int k;
   char *cp = new char('E');
   delete cp;
}
```

Obviously both the k and cp variables are local so they are allocated on the stack segment. Then when it comes the time to exit the function they will be freed from memory as the stack is restored.

The last statement, **delete cp;** is essential as it frees up the memory consumed in the function. If this statement is absent, the memory location pointed to by cp will no longer be accessible, once the control is returned out of the function. C++ does not take any responsibility to free such memory locations. The programmer has to take care of such things.

- **Careful with functions returning dynamic variables**

  Let us take a look at the following program.

```
#include <iostream>

char* tostring(int n) {
   char *S = new char[100];
   char aux;
   int i, j;

   for (i = 0; n; n /= 10, ++i)
      S[i] = n % 10 + '0';
   for (j = 0; j < i / 2; ++j) {
      aux = S[j];
      S[j] = S[i - j - 1];
      S[i - j - 1] = aux;
   }
   S[i] = '\0';

   return S;
}

void main() {
   cout << tostring(23) << tostring(146) << endl;
```

```
    char *temp;
    temp = tostring(23); cout << temp; delete []
temp;
    temp = tostring(146); cout << temp; delete []
temp;
}
```

Obviously the function char* tostring(int n) converts the integer n to a string, but that is not of our interest right now. You may have noticed that the string stored in S is not freed from the heap before exiting the function. We have just been warned about local pointers though. The reason for this is that the string should also be available within the calling function main() as we need to print it out to screen. To solve this "contradiction" we should first assign the return value to a temporary pointer variable inside main(), print it out and be sure to delete [] it right away, as shown above.

You may ask yourself why use a supplementary pointer here, why not stick to the previous variant which is also more compact ? The answer is simple - we may not be able to delete [] the dynamic variable returned by the tostring() function call as its address would eventually be lost if we do not store it somewhere. For example the calls tostring(23), tostring(146) within the cout statement return two dynamic variables whose adresses are only used at printing, they are then lost. This leads to memory leakage.

8.4. Classes and Dynamic Arrays

A dynamic array can have a base type, which is a class. A class can have a member variable, which is a dynamic array. The techniques of dynamic arrays and classes can be combined in many ways. The *string* class is an example

of such combination. The C++ string class contains a dynamic character array, that can be created to a desired size and can be initialized with a chosen value. Each object of the *string* class then represents a string. Other member functions to do basic operations on strings can be applied on these objects. The following C++ code defines and exercises a *string* class.

```
#ifndef STRVAR_H
#define STRVAR_H
#include <iostream.h>

class StringVar
{
public:
    StringVar(int size);
    //Initializes the object so it can accept string values up
to size
    //in length. Sets the value of the object equal to the
empty string.

    StringVar( );
    //Initializes the object so it can accept string values of
length 100
    //or less. Sets the value of the object equal to the
empty string.

    StringVar(const char a[]);
    //Precondition: The array a contains characters
terminated with '\0'.
    //Initializes the object so its value is the string stored in
a and
    //so that it can later be set to string values up to
strlen(a) in length

    StringVar(const StringVar& string_object);
    //Copy constructor.

    ~StringVar( );
```

```
    //Returns all the dynamic memory used by the object to
the heap.

    int length( ) const;
    //Returns the length of the current string value.

    void input_line(istream& ins);
    //Precondition: If ins is a file input stream, then ins has
    //already been connected to a file.
    //Action: The next text in the input stream ins, up to
'\n', is copied
    //to the calling object. If there is not sufficient room,
then only as
    //much as will fit is copied.

    friend ostream& operator <<(ostream& outs, const
StringVar& the_string);
    //Overloads the << operator
    //so it can be used to output values of type StringVar
    //Precondition: If outs is a file output stream, then outs
    //has already been connected to a file.
private:
    char *value; //pointer to the dynamic array that holds
the string value.
    int max_length; //declared max length of any string
value.
};

#endif //STRVAR_H
```

```
#include <iostream.h>
#include <stdlib.h>
#include <stddef.h>
#include <string.h>
#include "strvar.h"
```

```
//Uses stddef and stdlib.h:
StringVar::StringVar(int size)
{
   max_length = size;
   value = new char[max_length + 1];//+1 is for '\0'.
   if (value == NULL)
   {
      cout << "Error: Insufficient memory.\n";
      exit(1);
   }

   value[0] = '\0';
}

//Uses stddef and stdlib.h:
StringVar::StringVar( )
{
   max_length = 100;
   value = new char[max_length + 1];//+1 is for '\0'.
   if (value == NULL)
   {
      cout << "Error: Insufficient memory.\n";
      exit(1);
   }

   value[0] = '\0';
}

//Uses string.h, stddef, and stdlib.h:
StringVar::StringVar(const char a[])
{
   max_length = strlen(a);
   value = new char[max_length + 1];//+1 is for '\0'.
   if (value == NULL)
   {
      cout << "Error: Insufficient memory.\n";
      exit(1);
   }
```

```
    strcpy(value, a);
}


//Uses string.h, stddef.h, and stdlib.h:
StringVar::StringVar(const StringVar& string_object)
{
    max_length = string_object.length( );
    value = new char[max_length + 1];//+1 is for '\0'.
    if (value == NULL)
    {
        cout << "Error: Insufficient memory.\n";
        exit(1);
    }

    strcpy(value, string_object.value);
}


StringVar::~StringVar( )
{
    delete [] value;
}

//Uses string.h:
int StringVar::length( ) const
{
    return strlen(value);
}

//Uses iostream.h:
void StringVar::input_line(istream& ins)
{
    ins.getline(value, max_length + 1);
}

//Uses iostream.h:
ostream& operator <<(ostream& outs,
                     const StringVar& the_string)
```

```
{
   outs << the_string.value;
   return outs;
}
```

```
#include <iostream.h>
#include "strvar.h"


void conversation(int max_name_size);
//Carries on a conversation with the user.


int main( )
{
   conversation(30);
   cout << "End of demonstration.\n";
   return 0;
}

// This is only a demonstration function:
void conversation(int max_name_size)
{
   StringVar                    your_name(max_name_size),
our_name("Borg");

   cout << "What is your name?\n";
   your_name.input_line(cin);
   cout << "We are " << our_name << endl;
   cout << "We will meet again " << your_name << endl;
}
```

8.5. Copy Constructors and Destructors

A copy constructor is a special constructor that takes as its argument a reference to an object of the same class and creates a new object that is a copy. By default, the compiler provides a copy constructor that performs a member-by-member copy from the original object to the one being created. This is called a member wise or shallow copy. Although it may seem to be the desired behavior, in many cases a shallow copy is not satisfactory. For the *string* class, the default behavior of copy constructor is not is not sufficient for *string* class, as the data of the object is kept in a dynamic array. A member-by- member copy can only copy the address of the dynamic character array of the string object in to that of destination object. But the actual requirement is to copy the character sequence.

Hence the *string* class in the above listing redefines the default behavior of the copy constructor. In general notation the copy constructor can be written as A(A&) where A is the class name. The copy constructor of the string class is given below.

```
StringVar::StringVar(const StringVar& string_object)
{
   max_length = string_object.length( );
   value = new char[max_length + 1];   //+1 is for '\0'.
   if (value == NULL)
   {
      cout << "Error: Insufficient memory.\n";
      exit(1);
   }
   strcpy(value, string_object.value);
}
```

The copy constructor should be defined such that the object being initialized becomes a complete, independent copy of its argument. So in the above listing, a new dynamic character array is created, and character sequence is copied into it from the source array. A copy constructor is automatically called whenever C++ needs to make a copy of an object. Particularly in the following circumstances the copy constructor is called automatically:

1) When a class object is being defined and is initialized by another object of the same type.
2) When a function returns a value of the class type.
3) Whenever an argument of the class type is supplied for a call-by-value parameter.

If a class definition involves pointers and dynamically allocated memory using the new operator, then you need to include a copy constructor. Other classes do not need a copy constructor.

Destructors:

A destructor is a member function of a class that is called automatically when an object of the class goes out of scope. Destructors are used to eliminate any dynamic variables that have been created by the object so that the memory occupied by these dynamic variables is returned to the heap. The name of a destructor must consist of the tilde symbol ~ followed by the name of the class. A destructor takes no arguments and returns no value. The code given above for the string class also includes destructor for it. It contains the following lines.

```
StringVar::~StringVar( )
{
   delete [] value;
}
```

From the above function it is clear that the *stringvar* destructor deletes the dynamic character array value.

8.6. Summary:

Pointers are variables that hold addresses in C and C++. We can define a variable in C++ to store a *memory address*. A **pointer** in **C++** is said to "point to" the memory address that is stored in it. Also, when defining a ***C++*** pointer variable, we must specify the type of variable to which it is pointing. For example, to define a pointer, which will store a memory address at which exists an int, we can do the following: The other feature of c++ pointers is that they can be "re-seated", which means that you can change their value, you can change what they're pointing to, as in the following: // c++ pointer program for modifying values/re-seating.

An int* c++ pointer points to an int, so an int** points to an int. At that memory address exists a variable of type int*. This int* variable also stores a memory address, at which exists an int.

Arrays of built-in and user-defined data types may be dynamically allocated. User-defined data types include classes. We'll see dynamically allocated arrays of classes in a latter lesson, so for now let's look at built-in data types.

A copy constructor is a special constructor that takes as its argument a reference to an object of the same class and creates a new object that is a copy. By default, the compiler provides a copy constructor that performs a member-by-member copy from the original object to the one being created.

A destructor is a member function of a class that is called automatically when an object of the class goes out of scope. Destructors are used to eliminate any dynamic

variables that have been created by the object so that the memory occupied by these dynamic variables is returned to the heap.

8.7. Technical Terms:

Pointer: The memory address of a variable or object.

Pointer Variable: A variable that contains a memory address.

Constructor: A constructor is a method that has the same name as its class.

Destructor: A destructor is a method that has as its name the class name prefixed by a tilde, ~.

Copy Constructor: A copy constructor is a special constructor that takes as its argument a reference to an object of the same class and creates a new object that is a copy.

Exception: An exception is an error or anomaly that occurs as a program is executing. It can be due to a lack of system resources, such as a lack of memory or unavailability of a file, or raised by program design.

8.8. Model Questions:

1. What a Pointer? Explain different types of pointers?
2. What is a dynamic array? Explain how to create and use it?

3. Explain application of dynamic arrays with string class?
4. What is a copy constructor?
5. How do you delete dynamically allocated variables from memory in an object?

8.9. References:

Problem Solving With C++ by Walter Savitch, **Pearson Education Asia**

C++ by Balagurusamy, **BPB Publications.**

Let Us C++ by Y. Kanitkar.

AUTHOR:

Y. VENKATESWARA RAO, M.C.A.,
Lecturer,
Dept.Of Computer Science,
JKC College,
GUNTUR

Lesson 9 : Inheritance

Objectives

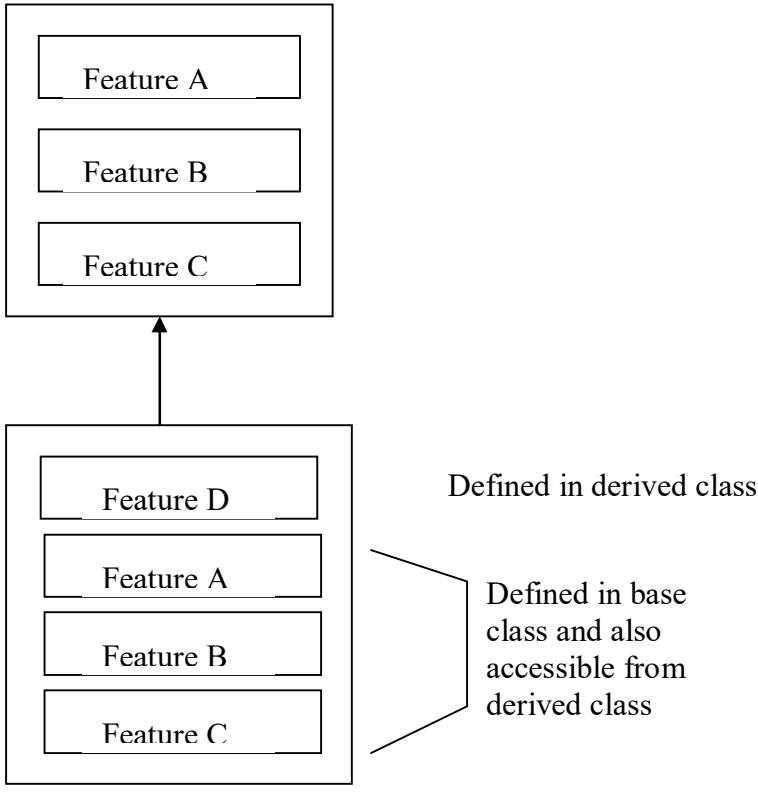
After completing this lesson you will understand about

- The concept of inheritance, and how it supports the concept of reusability.
- The derivation of a new class and the different visibility modes under which the new class is derived.
- The different types of inheritance.
- The execution of constructors in the derived class.
- The abstract class.

Structure Of The Lesson

9.1 Introduction To Inheritance

Inheritance is an important feature of object oriented programming, it is the process of extending a class to define new class. It allows new classes to be built from older and less specialized classes, instead of rewriting from the scratch. It allows the reuse of the pre-tested code of a class without changing the class. In this process the extended class is called base class and the newly defined class is called derived class. The base class is sometimes called a super class and correspondingly derived class is called sub class.



The derived class inherits all the capabilities of the base class and can add refinements and extensions of its own. There is no specific limit on the number of classes that may be derived from one another, which forms a class hierarchy.

C++ supports the access specifiers private, public and protected. As far as access limit is concerned, the private and protected members can be accessed only within the class . Public members are always accessible directly by all users of the class.

```
Base class name
{
private:
:       // visible to member functions within its class but not in
derived class
:
protected
:       //visible to member functions within its class and derived
class
:
public
:       // visible to member functions within its class,
:       //derived class and through object
};
```

9.2 Derived Class Declaration

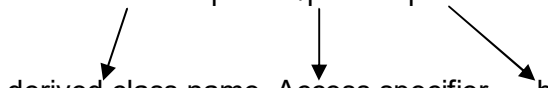
The derived class extends its features by inheriting the properties of another class, called base class and adding features of its own. The declaration of derived class specifies its relationship with the base class in addition to its own features.

The syntax for declaring a derived class is:

```
class derivedclass : [visibility mode Or Access specifier] baseclass
{
//Derived class members  (member functions and variables)
};
```

In the above syntax **class** is a keyword, **derived class** is name of the new class. "**:**" is used as a separator between the derived class name and the access speciifier. The visibility mode tells the way in which the base class is inherited (private, protected or public). The visibility mode is optional and the default mode is private. "baseclass" is the name of the class from which the properties are being inherited.

e.g.: class student : private/public person



derived class name Access specifier base class

{

…… };

Visibility Modes: There are three types of visibility modes. They are:

i) Private
ii) Protected
iii) Public

These are used for specifying the way in which the properties of the base class are inherited.

Private: If the access specifier "private", is used to inherit the properties of base class, then

i) The private data of the base class cannot be inherited but can be accessed through the inherited member.

ii) The protected data of the base class is inherited as the private data. It cannot be used in the main function. But, it can be accessed using the base class or the derived class member function.

iii) The public data of the base class is inherited as private data in the derived class. They are inaccessible to the objects of the derived class. It can be accessed by the member functions of the derived class.

e.g:

```
class base
{
private:
int x;
readx();
protected:
int y;
ready();
public:
int z;
readz();
}

class der: private base
{
private:
```

```
        int w;
public:
        void read();
void display();
};
```

In the above example, variable x cannot be inherited, but both y and z are inherited as private variables in the derived class. They can be accessed through the functions read() and display(). They cannot be used directly by the main().

Protected: When the access specifier “protected” is used to inherit base class properties,

- The private data of the base class cannot be inherited but can be accessed through the inherited members.
- Protected member in the base class are inherited as protected data in the derived class.
- The public data in the base class is inherited as protected data of the derived class.

```
class base
{
private:
int x;
readx();
protected:
int y;
ready();
public:
int z;
readz();
}
class der: protected base
{
private:
        int w;
public:
        void read();
        void display();
};
```

In the above example, ‘x’ cannot be inherited. y and z are inherited as protected members and thus can be used in read() and display() they can be further inherited but they can not be accessed from the ,main

Public: If the access specifier "public" is used to inherit the properties of base class, then

- The private data of the base class cannot be inherited as member of derived class but can be accessed through the inherited member functions.
- The protected member of the base class is the member of the derived class.
- The public member of the base class is the public member in the derived class.

```
class base
{
private:
int x;
readx();
protected:
int y;
ready();
public:
int z;
readz();
}
class der: public base
{
private:
        int w;
public:
        void read();
        void display();
};
```

In the above example 'x' cannot be inherited into the derived class, 'y' is inherited as a protected member of the derived class and 'z' can be accessed from the main itself.

Visibility Of Inherited Members:

Base class visibility	Derived class visibility		
	Public derivation	Private derivation	Protected derivation
Private	Not inherited	Not inherited	Not inherited
Protected	Protected	Private	Protected
Public	Public	Private	Protected

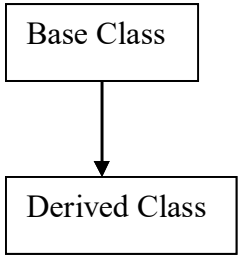
9.3 Types Of Inheritance

There are five types of inheritance

a. Single inheritance
b. Multilevel inheritance
c. Hierarchical inheritance
d. Multiple inheritance
e. Hybrid inheritance

9.3.1 Single Inheritance

Derivation of a class from only one base class is called single inheritance.



Programs To Illustrate Simple Inheritance

```
class A
{
      protected:
            int a_data;
      public:
            A ( ) {a_data =0 ;}
            A (int X)       {a_data =X ;}
            void showA ( )
            {
                  cout<< "\n\t a_data;   }
};

class B: public A
{
     private:
          int    b_data;
     public:
          B ( ) {b_data = 0 ;}
```

```
        B (int X) {b_data =X ;}
        void print _total ( )
        {
            cout<< "\n\t total ="
                << a_data +b_data;
        }
        B (int X, int Y)
        {
           a_data = X;
           b_data = Y;
        }
      };
void main ( )
{
        B    bobj (10, 20);
        bobj. print_total ( );
}
```

Program to display salaried employees data using single inheritance.

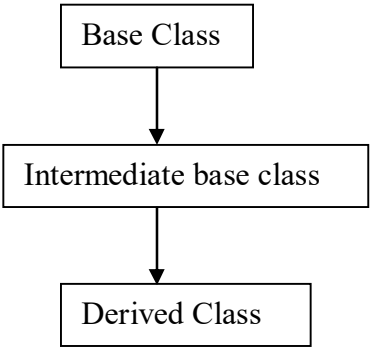
```
# include < iostream. h>
# include < string. h>
  class employee
   {
      public:
       employee (char * name, char * id, char * addr);
       void print ( );
   protected:
       char name [25], id [20], addr [100];
        double net_Sal;
   };
 class salaried_employee: public employee
 {
        public:
            salaried_employee (char * name, char *id,
char * addr, double sal);
            void show ( );
            void give rise ( );
        private:
            double basic_pay;
 };
employee:: employee (char * na, char * id, char * addr)
{
        strcpy (name, na);
```

```
            strcpy (id, idy);
            strcpy (addr, adr);
             net_sal =0;
}
void employee: : print ( )
{
        cout<< " \n\t Name:" <<name
              << "\t    ID:" <<id;
        cout<< "\n\t Address:"  << addr;
}
salaried_employee:: salaried_employee (char *na, char
*idy, char * addr, double sal): employee (na, idy, addr)
{
         basic_pay =sal;
}
          void salaried employee ::give_raise ( )
{
        basic_pay  += 100;
}
void salaried employee : : show ( )
{
        print ( );
        cout<< "\n\t  Salary:" << basic_pay;
}
void main ( )
{
        salaried Employee    e1 ( "Ravi", "ID 1", " unknown",
5000);
        e1. show ( );
        e1.  give_raise ( );
        e1. show ( );
}
```

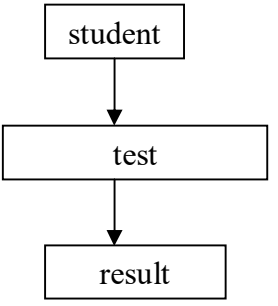
9.3.2 Multilevel Inheritance

When a class is derived from another derived class, such type of inheritance is known as Multilevel Inheritance.

A derived class with multilevel inheritance is as follows:



Program to display the result of the student using multilevel inheritance




```
#include<iostream.h>
class student
{
protected:
int rno;
public:
void getnum(int x)
{rno = x;}
void putnumb()
{ cout<<"Roll number:"<<rno<<endl;
}
};
class test: public student
{
protected:
float m1,m2;
public:
```

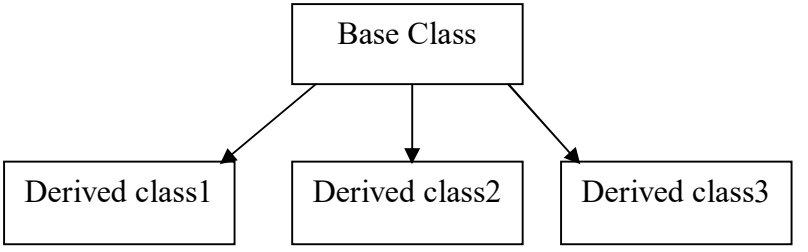
```
void getmarks(float x,float y)
{
m1 = x;
m2 = y;
}
void putmarks()
{
cout<<"Marks in sub1="<<m1<<endl;
cout<<"Marks in sub2="<<m2<<endl;
}
};

class result: public test
{
float total;
public:
void display()
{
total = m1 + m2;
putnumb();
putmarks();
cout<<"Total ="<<total<<endl;
}
};
void main()
{
result st1;
cout<<"STUDENT INFORMATION"<<endl<<endl;
st1.getnum(111);
st1.getmarks(78.0,89.5);
st1.display();
}
```

output:

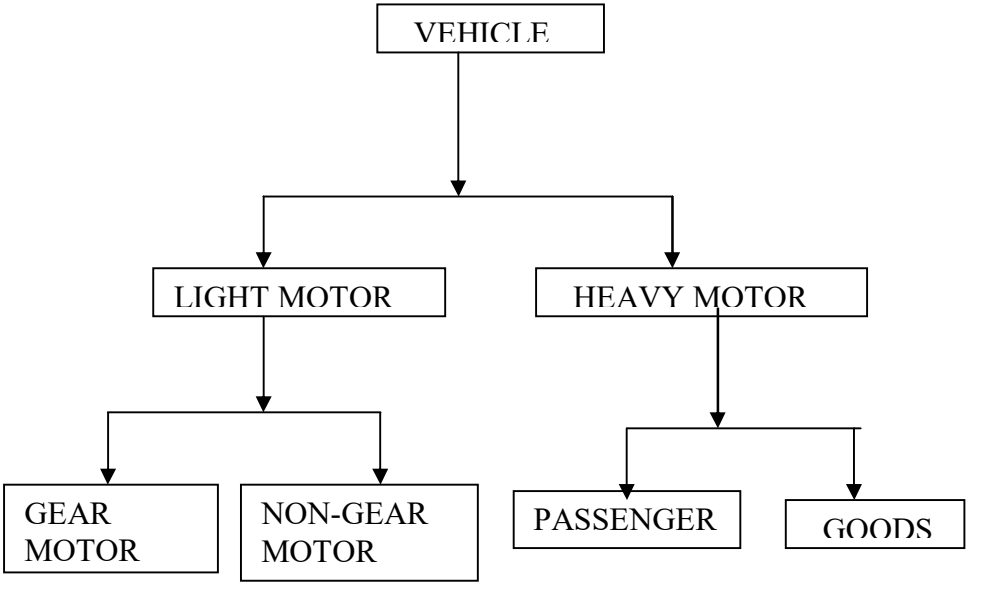
```
STUDENT INFORMATION
Roll number:111
Marks in sub1=78
Marks in sub2=89.5
Total =167.5
```

9.3.3 Hierarchical Inheritance

Derivation of several classes from a single base class i.e., the traits of one class may be inherited by more than one class, is called hierarchical inheritance.



Here is an example for hierarchical inheritance:




```
# include<iostream.h>
# define MAX_LEN  25
class vehicle
{
        protected:
```

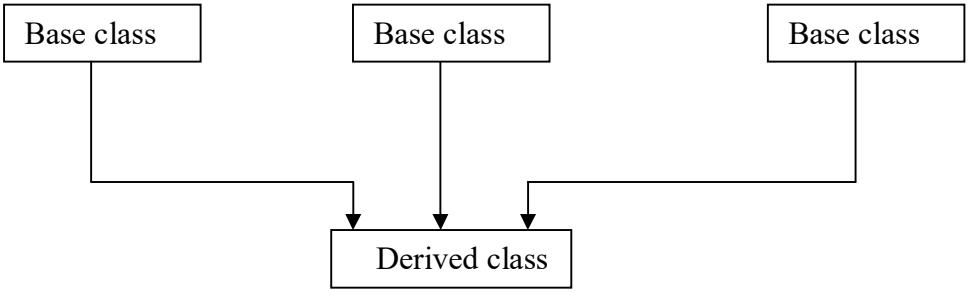
```
                    char  name[MAX_LEN];
                     unsigned wheelscount;
          public:
               void getdata( )
                {
                      cout<< " \n\t Name?";
                      cin>>name;
                      cout<< "\t Wheels?";
                      cin >> wheelscount;
}
void displaydata ( )
{
                    cout<< " \n\t Name:"
                          << name
                          << " \t Wheels :"
                           << wheelscount;
 }
};
class lightmotor : public vehicle
          {
             protected:
                int speedlimit;
             public:
                void getdata( )
                {
                          vehicle :: getdata();
                           cout<< "\n\t speed?";
                            cin>>speedlimit ;
                 }
                void displaydata()
                 {
                            vehicle :: displaydata( );
                             cout << "\n\t speed limit
:"<<speedlimit;
                  }
          };
class heavymotor : public vehicle
{
          protected:
                  int loadcapacity;
                   char permit[MAX_LEN];
           public:
                   void getdata ( )
                   {
                           vehicle :: getdata( );
```

```
                cout<< " \t load?";
                cin >>loadcapacity;
                cout<< " \t Permit?";
                cin >>permit;
            }
            void displaydata( )
            {
                vehicle :: displaydata( );
                cout << "\n\t Load :"
                    << loadcapacity
                    << " \n\t Permit:"
                    << permit;
            }
};
class gearmotor: public lightmotor
{
    protected:
        int gearcount;
    public:
        void getdata( )
        {
            lightmotor :: getdata( );
            cout<< " \t No of gears ?";
            cin>> gearcount;
        }
        void displaydata( )
        {
            lightmotor :: displaydata( );
            cout << " \t gear :" << gearcount;
        }
};
class nongearmotor : public lightmotor
{ };
class passenger : public heavymotor
{
    protected:
        int sitting, standing;
    public :
        void getdata( )
        {
            heavymotor ::getdata( );
            cout<< "\n\t Seats?";
            cin >> sitting;
            cout << "\t  standing  ?";
            cin >> standing;
```

```
                }
                void displaydata( )
                {
                        heavymotor :: displaydata( );
                        cout<< "\n\t seats :" << sitting;
                        cout<< " \t standing :" << standing;
                 }
};
class goods :public heavymotor
{ };
void main ( )
{
     gearmotor  gm;
     nongearmotor  ngm;
     passenger p;
     goods g;
     cout<< "\t Input for Gearmotor :";
     gm.getdata( );
          cout<<"\n Gearmotar\n ";


       Seats?45
        standing  ?20

 Passenger carrier

       Name:bus       Wheels :6
       Load :600
       Permit:ap
       seats :45      standing :20
       input for goods carier:
       Name?lorry
       Wheels?6
       load?2000
       Permit?ap

 Goods Carrier

       Name:lorry     Wheels :6
       Load :2000
       Permit:ap
```

9.3.4 Multiple Inheritance

Derivation of a class from two or more base classes, this type of inheritance is called multiple inheritance. Multiple inheritance is shown in the following diagram.



Example: Let a class 'C' be derived from two base classes 'A' and 'B' then class C is defined with the following syntax

```
class c : public A, public B
{
  _ _ _ _
  _ _ _ _
  Body of class c
  _ _ _ _ _
  _ _ _ _ _ _
};
```

A program to demonstrate multiple inheritance

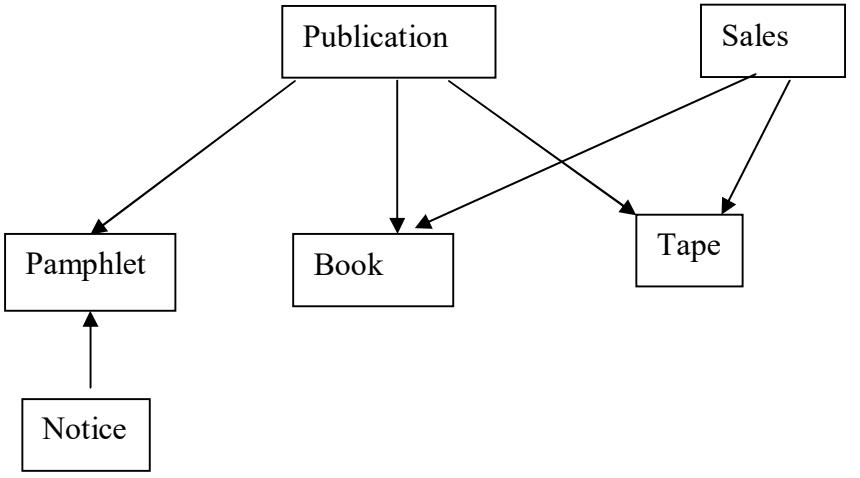
```
#include<iostream.h>
class A
{
      protected:
            int  a;
       public:
             A( ) { a = 0;}
             A( int d) {a= d;}
};
```

```
class B
{
        protected :
            int b;
         public:
            B ( ) {b=0;}
            B( int x) {b = x;}
};
class C: public A,public B
{
       private:
              int  c;
       public:
              C( ) {c = 0;}
              C(int i, int j, int k): A(i), B(j)
               {
                    c =k;
               }
               void display( )
               {
                        cout<< "\n\t sum =" <<(a+b+c);
                }
};
void main ( )
{
            C    cobj (10,20,30);
                 cobj.display ( );
}
```

output:

```
sum = 60
```

Program related to products company modeling using multiple inheritance




```
#include<iostream.h>
class publication
{
        protected :
             char title [50];
             float price;
           public:
              void getdata( )
              {
                    cout<< "\n\t Title ?";
                                  cin>> title;
                     cout<< " \t price ?";
                     cin >> price;
              }
              void putdata( )
              {
                   cout<< " \n\t Title:"   << title;
                   cout<< " \t price:"  <<price;
              }
};
class sales
{
          protected :
              float publishsales[3];
```

```
        public:
            void getdata( )
            {
                int i;
                for (i=0; i<3; i++)
                {
                    cout<< "Sales of the month" <<(i+1)
                        << " :";
                    cin>> publishsales[i];

                }
                                                    }
             void putdata( )
             {
                  int i; float tot = 0;
                  for(i =0; i<3; i++)
                  {
                       cout<< " \n sales of the month"<<(i+1)
                            << ":"<< publishsales[i];
                        tot  += publishsales[i];
                   }
                   cout<< " \n\t Total sales :"<<tot;
             }
};
class  book : public publication, public sales
{
      private:
            int   pages;
       public:
            void getdata( )
            {
                 publication :: getdata( );
                  sales :: getdata( );
                 cout<< "\n\t #of pages ?";
                  cin >> pages;
             }
             void putdata( )
             {
                publication :: putdata( );
                 sales ::putdata( );
                 cout<< "\n\t No of pages :"<<pages;
              }
};
class tape: public publication, public sales
{
```

```
        private:
              float playtime;
        public:
            void getdata( )
             {
                 publication ::getdata( );
                   sales ::getdata( );
                   cout<< " \t play time ?";
                    cin >> playtime;
              }
               void putdata( )
              {
                       publication :: putdata( );
                       sales :: putdata( );
                        cout<< " \n\t play time:"
                              << playtime;
              }
};
class pamphlet : public publication
{ };
 class notice : public pamphlet
{
        private:
             char  whom[20];
        public :
              void getdata( )
              {
                   pamphlet ::getdata( );
                   cout<< " \t To whom ?";
                    cin>> whom ;
              }
              void putdata( )
              {
                        pamphlet ::putdata();
                   cout<< " \t To whom :?"<< whom ;
                              }
 };
void main()
          {
                   book      book1;
                   tape       tape1;
                   pamphlet    p1;

                   cout<< "\n\t Enter Book data :";
                   book1.getdata( );
```

```
    cout<< " \n\t Enter Tape Data :";
    tape1.getdata( );
          cout<< "\n\t Enter notice data :";
    notice1.getdata( );
    cout<< "\n\t \t OUTPUT \n";
    cout<< "\n Book:" ; book1.putdata( );
    cout<< "\n Tape :"; tape1.putdata( );
    cout << "\n Notice:";notice1.putdata( );
      } //end   of main ( )
```

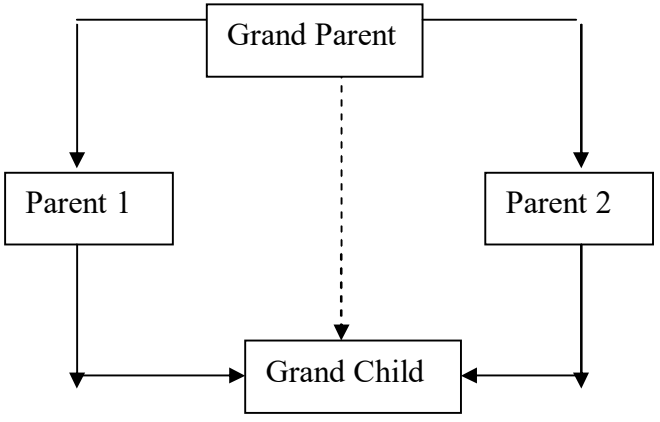
output:

```
          Enter Book data :
          Title ?c++
          price ?300
    Sales of the month1 :45
    Sales of the month2 :56
    Sales of the month3 :67


          #of pages ?450

          Enter Tape Data :
          Title ?Bhajans
          price ?50
    Sales of the month1 :60
    Sales of the month2 :70
    Sales of the month3 :80
          play time ?60

          Enter notice data :
          Title ?Games
          price ?10
          To whom ?students
          play time ?60


     OUTPUT

    Book:
          Title:c++       price:300
    sales of the month1:45
    sales of the month2:56
    sales of the month3:67
          Total sales :168
          No of pages :450
```

Tape :
Title:Bhajans price:50
sales of the month1:60
sales of the month2:70
sales of the month3:80
Total sales :210
play time:60
Notice: Title:Games price:10 To whom :?students

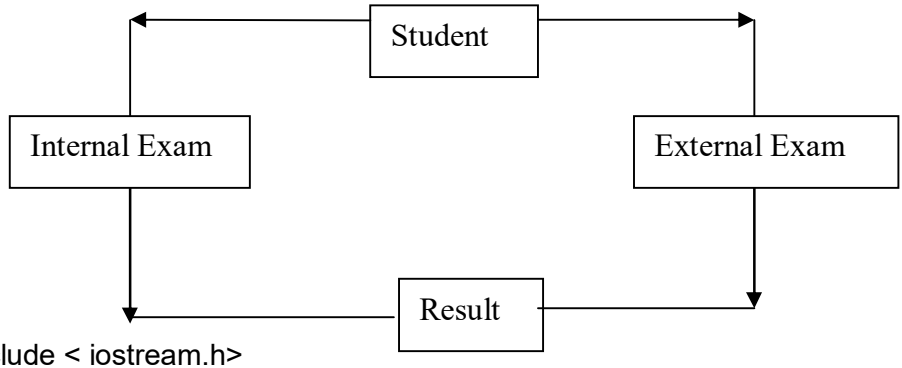
9.3.5 Multipath Inheritance

The form of inheritance, deriving a new class by multiple inheritance of base classes, which are derived earlier from the same base class, is known as multipath inheritance. This type of inheritance involves multiple forms of inheritance namely multilevel, multiple and hierarchical inheritance. The child class is derived from the same base class parent1 and parent2(multiple inheritance), which themselves have a common base class grandparent(hierarchical inheritance). The child inherits the properties of grandparent via two separate paths. The classes, parent1 and parent2 are referred as direct base classes, whereas grandparent is referred to as indirect base class. The inheritance diagram for multi-path inheritance is as shown below



Multipath inheritance may cause duplicate items to be derived into the child class, twice via parent1 and parent2, of the grandparent class. C++ avoids this ambiguity by introducing virtual base classes. This is done by making the common base class as a virtual base class, while declaring the direct or intermediate classes (parent1 & parent2).

The following example of student database uses multipath inheritance to derive the classes in the chart:




```
# include < iostream.h>
const int MAX_LEN =25;
class student
{
        protected:
              int     rollno;
              char  name [MAX_LEN];
        public:
               void read ( )
               {
                     cout<< "\n\t Roll_No ?";
                     cin >> rollno;
                     cout<< " \t name ?";
                     cin >> name;
               }
               void print ( )
               {
                      cout<< " \n\t Name:"
                           <<name << "\t Roll_No:"
                            <<rollno;
               }
};
class internalexam: public virtual student
{
             protected:
                   int marks1, marks2;
             public:
                    void readdata( )
                    {
                          cout<< "\n\t Input marks in2 subjects:";
                          cin>> marks1 >>marks2;
                     }
```

```
                    void displaydata( )
                    {
                        cout<< "\n\t Internal marks:"
                            <<marks1<< " "
                             <<marks2;
                        cout<< "\n\t\t Total:"
                             << gettotalinternal( );
                    }
                    int gettotalinternal( )
                    {
                            return (marks1+marks2);
                     }
};
class externalexam :public  virtual student
{
        protected:
            int marks1, marks2;
        public:
             void readdata( )
             {
                 cout<< "\n\t Input marks in 2 subjects:";
                 cin>> marks1 >>marks2;
             }
             void displaydata( )
             {
                 cout<< "\n\t External marks:"
                      << marks1<< " "  << marks2;
                 cout<< "\n\t\t Total:"
                       << gettotalexternal ( );
             }
 int gettotalexternal( )
  {
        return(marks1+marks2);
   }
};
class result:public internalexam, public externalexam
{
        public:
            int gettotalmarks( )
            {
                return (gettotalinternal( )+gettotalexternal( ) );
             }
};

void main ( )
```

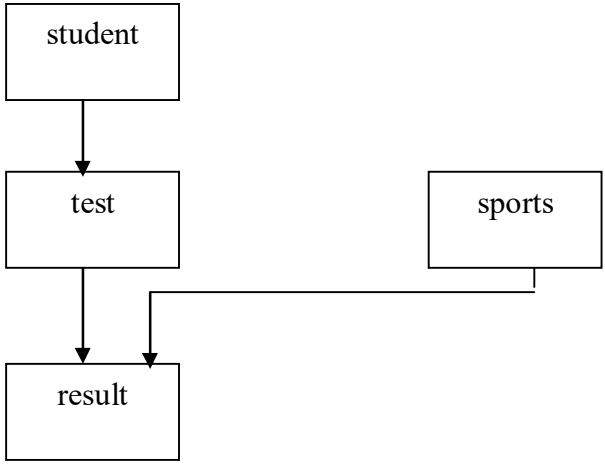
```
{
    result stud1;
    cout<< "\n\t Input student info:";
    stud1.read( );
    cout<< "\n\t Reading internal marks:";
    stud1.internalexam::readdata( );
    cout<< "\n\t Reading external marks:";
    stud1.externalexam ::readdata( );
    cout<<"\n\t\t OUTPUT:";
    stud1.print( );
    stud1.internalexam::displaydata ( );
    stud1.externalexam:: displaydata( );
    cout<<"\n\tTotal marks:"<< stud1.gettotalmarks();
}
```

output:

```
    Name:qwer       Roll_No:1
    Internal marks:23 24
          Total:47
    External marks:24 35
          Total:59
    Total marks:106
    Input student info:
    Roll_No ?1
    name ?sai

    Reading internal marks:
    Input marks in2 subjects:15  18

    Reading external marks:
    Input marks in 2 subjects:75  71

          OUTPUT:
    Name:sai       Roll_No:1
    Internal marks:15 18
          Total:33
    External marks:75 71
          Total:146
    Total marks:179
```

9.3.6 Hybrid Inheritance

Derivation of a class involving more than one form of inheritance is known as hybrid inheritance.



Program to calculate the result of the student using hybrid inheritance

```
#include<iostream.h>
class student
{
protected:
int rno;
public:
void getnum(int x)
{rno = x;}
void putnumb()
{ cout<<"Roll number:"<<rno<<endl;
}
};
class test: public student
{
protected:
float m1,m2;
public:
void getmarks(float x,float y)
{
m1 = x;
```

```
m2 = y;
}
void putmarks()
{
cout<<"Marks in sub1="<<m1<<endl;
cout<<"Marks in sub2="<<m2<<endl;
}
};
class sports
{
protected:
float score;
public:
void getscore(float s)
{score = s;}
void putscore()
{
cout<<"Sports score:"<<score<<endl;
}
};
class result: public test, public sports
{
float total;
public:
void display()
{
total = m1 + m2 + score;
putnumb();
putmarks();
putscore();
cout<<"Total ="<<total<<endl;
}
};
void main()
{
result st1;
cout<<"STUDENT INFORMATION"<<endl<<endl;
st1.getnum(111);
st1.getmarks(78.0,89.5);
st1.getscore(6.0);
st1.display();
}
```

output:

```
STUDENT INFORMATION
Roll number:111
Marks in sub1=78
Marks in sub2=89.5
Sports score:6
Total =173.5
```

9.4 Constructors In Derived Class

- As long as the base class has no constructor this derived class need not have constructor.
- If base class has the constructor with one or more arguments then it is compulsory for the derived class to have constructor and it must pass the arguments to the base class. The base class constructor is executed first then the derived.
- In multiple inheritance the base classes are constructed in the order in which they appear in the declaration of the derived class.
- In the multilevel inheritance the constructor will be executed in the order of inheritance.

Execution Of Base Class Constructor :

Method of Inheritance	Order of execution

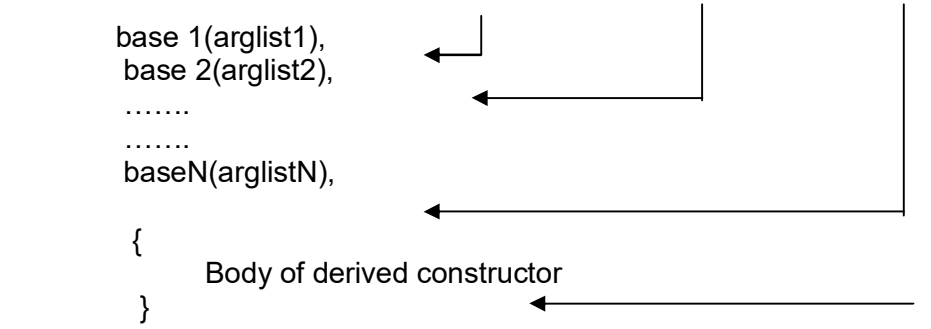
class B: public A
{
 	
};
	A( ): base constructor
	B( ): derived constructor

class A: public B , public C
{
};
	B( ) :base(first)
	C( ) :base(second)
	A( ) :derived

class A: public B, public virtual C
{
 	
};
	C( ) :virtual base
	B( ) :ordinary base
	A( ) :derived

The General Form Of Defining A Derived Constructor Is:

Derived-constructor (Arglist 1, Arglilst 2, …. ArglistN , Arglist (D)



A sample program using derived class constructors

```
#include<iostream.h>
class alpha
{
        int x;
         public :
            alpha ( int i )
             {
               x=i;
               cout<<"Alpha is initialized\n ";
             }
            void show_x( )
             {
                cout<<"x="<<x<<endl;
              }
};
class beta
{
       float y;
        public :
           beta ( float j)
                              {
                                        y = j;
                                        cout<<"Beta initialized\n";
                              }
```

```
                void show_y(void)
                        {
                                cout<<"y="<< y<<"\n";


                        }
};
class gamma:public beta,public alpha
{
        int m,n;
        public:
                gamma(int a, float b,int c, int d): alpha(a),beta(b)
        {
                m = c;
                n = d;
                cout<<"Gamma initialized\n";
        }
                void show_mn(void)
        {
                cout<<"m= "<<m<<"\nn="<<n<<"\n";
        }
};
void main()
{
        gamma g(5,10.75,20,30);
        cout<<"\n";
        g.show_x();
        g.show_y();
        g.show_mn();
}
```

output:

```
        Beta initialized
        Alpha is initialized
        Gamma initialized

        x=5
        y=10.75
        m= 20
         n=30
```

9.5 Abstract Class

An abstract class is a class which is not used to create objects. It acts as only a base class for the other derived class. A class that contains at least one pure function is known as abstract class.

9.6 Summary

- We have covered the concept of inheritance. The derivation of a new class from the base class.
- The different visibility modes and the visibility of the base class inheritance in different modes are studied.
- The different types of inheritance are covered in detail.
- The usage of constructors in the derived class is also covered.

9.7 Technical Terms

Abstract class: A class that serves as only base class from which classes are derived. No object of abstract base class are created.

Base class: A class from which other classes are derived.

Derived class: A class that inherits some or all of its functions from another class, called the base class.

Intermediate class: A class that lies on an inheritance path connecting a base class and a derived class.

Protected member: A protected member is same as a private member except that protected members of the base class can be inherited.

Reusability: A feature of OOP where it allows the reuse of existing class without redifinition.

Virtual base class: A base class that can be qualified as virtual in the inheritance definition. For a virtual base class, only one copy of its members can be inherited regardless of the number of inheritance paths between the base class and the derived class.

Visibility : The ability of one object to be server to others.

9.8 Model Questions

1. What is inheritance? Explain with example.
2. What is the difference between base and derived classes?
3. What are the different forms of inheritance supported by C++? Explain with examples.
4. Explain the importance of Inheritance.

9.9 References

Object-oriented programming with C++,
by E. Bala Guruswamy.

Problem solving with C++
by Walter Savitch

Mastering C++
by K.R.Venugopal, RajkumarBuyya, T.RaviShankar

AUTHOR:

M. NIRUPAMA BHAT, MCA., M.Phil.,
Lecturer,
Dept. Of Computer Science,
JKC College,
GUNTUR.

Lesson : 10 Virtual Functions And Templates

Objectives

After completing the lesson you will understand about:

- Types of polymorphism
- Defining the pointer to objects and functions
- Overriding the base class functions by virtual functions
- Generic programming using function templates and class templates

Structure Of The Lesson

10.1 Introduction To Polymorphism And Virtual functions

Polymorphism means “one name with multiple forms”. The process of resolving or mapping a function call with function definition is known as “binding”. In general, the function calls are resolved or mapped to function definitions when the program is complied. This is called “compile time binding” or “compile time polymorphism” . When virtual functions are called on objects they are not resolved at compile time, and postponed till the execution of program. Hence it is called “late binding” or “dynamic binding”.

Dynamic binding is one of the powerful features of C++. This process uses the pointer to objects. We will see how pointers to objects and virtual functions are used to implement dynamic binding.

10.2 Pointers To Objects

A pointer can point to an object created by a class.
e.g:

```
class item
{
    int code;
    float price ;
        public:
        void getdata(int,float);
        void show();
 }
item i1;
item * ptr;
ptr=&i1;
 ptr= new item;
```

“**item** “ is a class with int code and float price. **getdata()** and **show(**) are its member functions. **i1** is an object of type item. ***ptr** is a pointer pointing to the type item. The item i1 is accessed by **i1.getdata()** . It can also be accessed using the pointer variable pointing to that function i.e., **ptr->getdata()**. When the statement,

```
ptr = new item;
```

is executed it allocates enough memory for the data members in the object structure and assigns the address of member space to the pointer ptr.

// A sample program to access the object through pointers

```
#include<iostream.h>
        class item
        {
                int code;
                float  price ;
                 public  :
                        void  getdata (int  a, float b )
                         {
                                code =a;
                                 price = b;
                         }
                        void show ( )
                        {
                                cout<<"code:"<< code <<endl;
                                 cout<<"price:"<<price<<endl;
                         }
        };
void main ( )
{
        item  *p= new item[2];
        item   *d = p;
         int x, i ;
        float y;
         for ( i =0 ;    i <2 ; i++ )
            {
                cout<<"Enter code & item \n:";
                 cin>>x>>y;
                 p-> getdata ( x, y );
                  p++;
             }
            p=d;
            for ( i= 0 ; i<2 ; i++)
               {
                      cout << " item:\n";
```

```
                p->show ( );
                p++;                }
        d= NULL;
        delete [ ] p;

}
```

output:

```
Enter code & item
:1
2.2
Enter code & item
:3
3.4
 item:
code:1
price:2.2
 item:
code:3
price:3.4
```

10.3 Pointers To The Derived Class

Pointers can be used to point to the base class as well as the derived class objects. A single pointer can point to different classes.

For e.g.:

```
Let B be the base class and D is the derived class.
B * bptr ; // pointer to the base class
B       b ;  // Base class object
D       d ; // Derived class  object
bptr = &b; // pointer points to the  object of the base class B
bptr =&d; // pointer points to the object of the derived class D.
```

By pointing the base class pointer to the derived, the member inherited by the base class can be accessed but the members of derived class cannot be accessed.

Program to demonstrate the baseclass pointer pointing to the derived class

```
# include <iostream.h>
class base
{
  public : int b;

   void show( )
    {
        cout<<"This is a base class \n";
           cout<<"b= "<<b<<"\n";
     }

};

class derived : public base
  {
       public : int d;

        void show( )
         {
            cout<< " This is derived ";
            cout << "b = " <<b;
            cout<< "d = "<<d;
         }
};

void main ( )
{
    base * bptr;
    base bs;
    bptr = & bs;
    bptr -> b = 100;
    bptr -> show ( );

    derived  der ;
    bptr = &der;
     bptr-> b =50;
     //bptr->d = 300       //doesnot work
    bptr->show();

    derived  * dptr;
           dptr = &der;
```

```
dptr -> d = 70;
dptr -> show ( );  }
```

output:

```
This is a base class
b= 100
This is a base class
b= 50
 This is derived class b = 50d = 70
```

10.4 Overriding The Base Class Functions Using Virtual Function

When a derived class redefines a base class member function the member function call on the objects uses the definition given in derived class. But whether the same member function is called on the derived object using a base class type pointer it leads to the function definition in the base class to be executed instead of the definition given in derived class. The compiler ignores the contents of the pointer and chooses the member function that matches the pointer. To avoid this, the base class version of the function should be declared as a “virtual” function. This is done by, inserting the keyword “**virtual**” to the function declaration in base class. The usage of virtual functions leads to late binding of function calls i.e., function calls to the virtual functions are resolved at the time of running the program. This is also called “execution time binding” or “late binding” or “runtime polymorphism”.

When the function, the function executed at the runtime is based on the type of the object pointed to, by the base pointer rather than the type of the pointer.

Note: Coercion is the automatic conversion of one type of value to another type.

Program to demonstrate virtual functions

```
#include<iostream.h>

class bc
{
public:
 void  display()
 {
```

```
 cout<<"Display base\n";
 }
  void virtual show()
 {
 cout<<"This is base\n";
  }
};
class dc :public bc
{
public:
 void display()
 {
 cout<<"Display derived\n";
 }
 void show()
 {
 cout<<"This is derived \n";
 }
};

void main()
{
bc *bptr;
bc bas;
cout<<”Baseclass pointer is pointing to baseclass
object\n”;
bptr = &bas;
bptr->display();
bptr->show();
dc *dptr;
dc der;
cout<<”Baseclass pointer is pointing to derived class
object\n”;
bptr = &der;
bptr->display();
bptr->show();
cout<<”Derived class pointer is pointing to derived class
object\n”;
dptr = &der;
dptr->display();
dptr->show();
}
```

output:

```
Baseclass pointer is pointing to baseclass object
Display base            //Base class functions are executed
This is base
Baseclass pointer is pointing to derived class object
Display base            //Baseclass function is executed
This is derived//As it is virtual, derived class function is executed
Derived class pointer is pointing to derived class object
Display derived         //Derived class functions are executed
This is derived
```

Program To Demonstrate Hierarchical Inheritance Using Virtual Functions:

```
//Gshape class provides basic functionality for any geometric shape. It
//contains a virtual function area, which is overridden in each of the
//derived classes.
    # include < iostream.h>
    #include<conio.h>
    class Gshape
    {
        public:
           virtual double area ( ) {return 0;}
    };
    class Triangle : public Gshape
    {
        double base, height;
        public:
            Triangle ( ) {base = height>0.0;}
             Triangle (double  b, double h)
             {
                  base = b; height =h;
              }
              double area ( )
              {
              return (0.5 * base * height ) ;
               }
    };
    class Circle : public Gshape
    {
            double radius;
        public :
            Circle ( ) {radius =0.0 ;}
             Circle (double r)
```

```
                {
                    radius =r;
                }
                double area ( ) ;
};
class Rectangle : public Gshape
{
                double length, breadth;
                 public:
                      Rectangle ( ) {length = breadth =0.0;}
                      Rectangle (double l, double b)
                       {
                           length =l, breadth =b;
                        }
                      double area ( );
};
double Circle ::area ( )
{
          return (3.141 * radius * radius );
 }
double Rectangle :: area ( )
{
          return (length * breadth );
}
void main ( )
{

                Triangle t(5.5, 3.5 );
                Circle     c(7.0);
                 Rectangle   r (8.0, 9.5);
              Gshape * objects [3];
                 objects[0] = &t;
                 objects[1] =&c;
                 objects[2] = &r;
                                          clrscr();
    cout<<"area of triangle, circle,rectangle are_ _ _";
                    for (int i=0; i<3; i++ )
                         cout << "\t"  << objects [i] ->area ( );
}
```

output:

area of triangle, circle,rectangle are_ _ _ 9.625 153.909 76

10.5 Templates

Templates are an important feature of C++. It provides greater flexibility to the language by supporting generic programming. This allows the reusable of software components such as functions, classes, etc., supporting different data type in a single framework.
A template is like a blueprint, which can be used to create a number of similar objects. It allows the construction of family of template functions and classes to perform the same operation on different data type. The templates declared for functions are called function templates and those declared for class are called class templates.

10.5.1 Function Templates

A function template in C++ is a sub program defined using a generic algorithm and generic data types. A generic algorithm is an algorithm that can be applied to different types of data to achieve similar results. In C++ we use function templates to generate similar functions for different data types. A generic function defines the general set of operation that will be applied to various types of data. It has the type of data that will operate upon when passed as a parameter. The same general procedure can be applied to a wide range of data. The generic function is independent of any data. The compiler automatically generates the correct code for the type of data that is actually used when the function is executed.

The definition of the template begins with " template <class T>" it is called template prefix. It tells the compiler that the definition or prototype that follows this statement is a template. "T" is a type parameter. The word class means the type or data type. The type parameter 'T' can be replaced by any type i.e., it may be simple data type or user defined data type. Within the body of the function definition the type parameter "T" is used just like any other data type.

The Syntax For Defining Function Templates:

```
template <class    type_ parameter>
Return type function name (parameter ;list);
```

The parameter list should contain at least one parameter of generic type.

e.g.:

```
Template<class  T>
Return type      void  swap(T&V1, T&V2);
```

The definition of a template function is written as any other function definition. The function header will begin with the template prefix. The template function can be called from the main program like any function call.

Consider swapping of two variables. The swapping algorithm is same for any type of variables. In a generic type a swap function can be defined as follows:

e.g.:

```
template<class T>
void swap(T&a,T&b)
 {
T    t;
 t = a;
 a= b;
 b= t;
 }
```

The above function template can be applied to swap values of two integer variables by replacing “T” with “int”. The same templates can also be applied to swap the values of two objects of a given type (class) by substituting the class name for “T”(variable type). Like this the above function template can be used to swap values of any type of variables.

Function template is used with different types of data, as the complier will produce a separate definition for each different type that we use with the function template. The function call statement for template function is written like any other function.

For Example:

```
swap (ch1, ch2);          // ch1 & ch2 are character variables
swap(f1, f2);             // f1&f2  are floating point variables
```

Program To Swap The Contents Of Two Variables Using A Template Function

```
#include<iostream.h>
template<class T>
void swap(T& var1, T& var2)
{ T temp;
temp = var1;
var1 =var2;
var2 =temp;
}
int main()
{
int int1 = 1, int2 =2;
cout<<"Integer values are "<<int1<<" and "<<int2<<endl;
swap(int1,int2);
cout<<"Integer values after swapping are "<<int1<<" and "
<<int2<<endl;
double d1 = 1.1, d2 =2.2;
cout<<"Double  values are "<<d1<<" and "<<d2<<endl;
swap(d1,d2);
cout<<"Double values after swapping are "<<d1<<" and" << d2
<<endl;
char symb1 = 'A', symb2 ='B';
cout<<"Character values are "<<symb1<<" and "<<symb2<<endl;
swap(symb1,symb2);
cout<<"Character values after swapping are "<<symb1<<" and
"<<symb2<<endl;
return 0;
}
```

output:

```
Integer values are 1 and 2
Integer values after swapping are 2 and 1
Double  values are 1.1 and 2.2
Double values after swapping are 2.2 and 1.1
Character values are A and B
Character values after swapping are B and A
```

Program For Swapping A List Of Values With Bubble Sort Technique Using Template Functions

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

template <class T >
```

```
void swap(T&v1,T&v2)
{
                T    t;
                t=v1;
                v1=v2;
                v2=t;
}
//generic function to display a list of values.
template<class T>
void showlist(T list[],int size)
{
        int i=0;
        for(;i<size;i++)
        cout<<'\t'<<list[i];
}

//template function to sort list of values
template<class T>
void bsort(T list[],int size)
{
          int i,j;
           for(i=0;i<size-1;i++)
             for(j=0;j<size-1;j++)
                  if(list[j]>list[j+1])
           swap(list[j],list[j+1]);
}
void main()
{
      float list1[]={10.f,-11.5f,13.0f,2.1f,8.4f};
                int list2[]={3,11,5,2,1};
      cout<<"\n\t floats before sort"<<endl;
      showlist(list1,5);
      bsort(list1,5);
      cout<<"\n\t floats after sort"<<endl;
      showlist(list1,5);
      cout<<"\n\tintegers before sort"<<endl;
      showlist(list2,5);
                bsort(list2,5);
      cout<<"\n\tintegers after sort"<<endl;
      showlist(list2,5);
}//main()
```

output:

floats before sort

```
10      -11.5   13      2.1     8.4
 floats after sort
-11.5   2.1     8.4     10      13
integers before sort
3       11      5       2       1
integers after sort
1       2       3       5       11
```

Template Function For Binary Search:

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

//template function for show_list()
template<class T>
void showlist(T list[],int size)
{
            for(int i=0;i<size;i++)
                  cout<<"\t"<<list[i];
}
//template function for binary search
template<class T>
void bisearch(T list[], T se,int low,int high,int& pos)
{
         int mid;
          if(low<=high)
          {
                 mid=(low+high)/2;
                  if(list[mid]==se)
                                                          pos=mid;
                   else if(list[mid]< se)
                   {
                               low=mid+1;
                                   bisearch(list,se,low,high,pos);
                    }
                    else           if(list[mid]>se)
                    {
                                 high=mid-1;
                                 bisearch(list,se,low,high,pos);
                     }

          }
}
```

```
void main()
{
          int pos=-1;
           float    list1[]={-11.5f,10.2f,13.5f,22.6f};
                              int list2[] = {4,12,26,31};
             cout<<"\n\t float type data";
              showlist(list1,4);
              bisearch(list1,22.6f,0,3,pos);
              cout<<"\n\t element found at:"<<pos+1;
                                        cout<<"\n\t integer data";
              showlist(list2,4);
              bisearch(list2,26,0,3,pos);
            cout<<"\n\t element found at:"<<pos+1;
}
```

output:

```
float type data       -11.5   10.2    13.5    22.6
element found at:4
integer data   4       12      26      31
element found at:3
```

10.5.2 Class Templates

Templates are also used in defining generic classes like generic algorithm. We can define generic classes and later substitute in the generic class to get more specific classes. A generic class is defined with type parameters. The syntax of class template starts with a template header and continues with the class definition.

Syntax for class templates:

```
template <class type parameters>
class class_name
{
          generic variables;
               .
                        .
               .
               .
         generic functions;
}
```

The generic class consists of generic member variables and functions. The class and member function definitions are same as for any ordinary classes expect the type parameter has to be used in place of data type. The member functions and overloading operators are defined as function templates. In the main program a type argument is given to the class template to make the object specific.

Program to demonstrate a class template for representing a pair of values.

```
#include<iostream.h>
template<class T>
class pair
{
        private:
          T   first,second;
        public:
        pair() { }
        pair (T a, T b)
         {
                first =a;
                 second=b;

          }
      void set (int pos, T  v);
       friend   ostream & operator<<(ostream & out,pair<T> p);
};
    //template  function for overloading stream operator.
   template<class T>
  ostream & operator<<(ostream&out,pair<T> p)
{
      out<<p.first<<"   "<<p.second;
       return out;
}
template < class T>
void pair<T>::set(int pos,T   v)
{
           if(pos==1)
                    first =v;
            else if(pos==2)
                     second=v;
}
void main()
{
         pair<int>      intpair;
```

```
                pair<char>     charpair('A','B');
                cout<<"\n\t The pair of characters:"<<charpair;
                 intpair.set(1,10);
                 intpair.set(2,20);
                 cout<<"\n\t The pair of integers:"<<intpair;
        }
```

output:

```
float type data        -11.5   10.2    13.5    22.6
element found at:4
integer data   4       12      26      31
element found at:3
The pair of characters:A   B
The pair of integers:10   20
```

Class templates are used for instantiation by substituting a data type name for a type parameter. This substitution creates a specific definition of the template class and leads to the creation of objects of that specific type, for example,"pair<int> int pair " creates int pair object to contain a pair of integers. This is because the data type int is substituted for the type parameter name 'T'. We can use "typedef" keyword to define types based on a specific data type substitution in the template class. For example

```
typedef pair<int> int pair;
```

defines 'int pair' as a class of pair of integers by substituting int for the type parameter 'T'. This data type can be used to instantiate objects for integer pairs i.e.,

```
int pair  p1;
```

declares p1 as an object that contains pair of integers.

A class template program to process a list of values

```
#include<iostream.h>
template<class T>
class list
{
        public:
                list ()
                   {     itemcount=0;}
                void  add(T ele);
                     int isfull ();
                 //return 1 if it is full otherwise 0;
                int length();
```

```
                friend ostream& operator<<(ostream& out,list<T> l);
        private:
                        T contents[20];
                         int itemcount;
};
template<class T>
 void list <T>::add(T ele)
{
                if(!isfull())
                {
                        contents[itemcount]=ele;
                         itemcount++;
                 }
                else
                        cout<<"\n  the list is full.";
}
template<class T>
ostream& operator<<(ostream& out,list<T> l)
{
               int i;
                for(i=0;i<l.itemcount;i++)
                        out<<l.contents[i]<<'\t';
                return out;
}
 template <class T>
int list<T>::isfull()
{
          if(itemcount==20)         return 1;
           else                              return 0;
 }
template<class T>
 int list<T>::length()
{
          return itemcount;
}

void main()
{
            list<int>li;
            list<char>lc;
            li.add(10);
            li.add(20);
            li.add(30);
            cout<<endl<<"The list of integers:"<<li;
            lc.add('A');
```

```
                    lc.add('B');
                     lc.add('C');
                     lc.add('D');
                     cout<<endl<<"The list of charactcers:"<<lc;
                     cout<<"\nThe total no of elements in both the
lists:"<<li.length()+lc.length();
        }
```

output:

```
        The list of integers:10 20  30
        The list of charactcers:A      B       C       D
        The total no of elements in both the lists:7
```

A program to demonstrate Inheritance Of Class Templates: To Prepare A Set Class And To Manipulate It.

```
#include<iostream.h>
#include<conio.h>
const int MAXSIZE=20;
template<class T>
class bag
{
        protected:
                T   contents [MAXSIZE];
                int itemcount;
        public:
                bag()
                        {
                        itemcount=0;
                        }
                int isempty()
                {
                   if(itemcount==0)
                                return   1;
                         else
                                return 0;
                 }
                 int  isfull()
                 {
                   if(itemcount==MAXSIZE)
                                return 1;
                         else
                                return 0;
                   }
```

```
                    void put(T item)
                    {
                      if(!isfull())
                         contents[itemcount++]=item;
                    }
                    int isexist(T ele);
                    void show();
};

template<class T>
 int bag<T>::isexist(T  item)
{
             int i;
             for(i=0;i<itemcount;i++)
                  if(contents[i]==item)
                     return 1;
              return  0;
}
template<class T>
void bag<T>::show()
{
            int i=0;
            for( ; i<itemcount;i++)
              cout<<" "<<contents[i];
              cout<<endl;
}
template<class T>
class set:    public bag<T>
   {
          public :
                 void add(T item);
                 void read();
                 void operator=(set <T> s);
                 friend set <T> operator +(set  <T> s1, set<T> s2);
    };
template<class T>
 void set<T>::add(T item)
         {
         if(!isexist(item) && !isfull())
                  put(item);
}
template<class T>
 void set<T>::read()
{
         T item;
```

```
            char   ch;
              while(1)
               {
                     cout<<"\n\t input an element:";
                     cin>>item; add(item);
                     cout<<"\t\t another (Y/N) ?";
                     cin>>ch;
                     if((ch=='N' )|| (ch=='n'))
                       break;
                } //end of while statement.
} //end of function.

template<class T>
void set<T>::operator=(set<T> s)
{
            int i;
            itemcount= s.itemcount;
            for(i=0;i<itemcount;i++)
              contents[i]=s.contents[i];
}
template<class T>
set<T> operator +(set <T> s1, set<T> s2)
{
          set <T> temp;
          temp=s1;
           for(int i=0;i<s2.itemcount;i++)
                 temp.add(s2.contents[i]);

                     return   temp;
}

void main()
{
            set <int>  s1;
                              set <int> s2;
                              set <int> s3;
            set <char> s4;
             cout<<"\n\t  enter  integers for set1";
              s1.read();
             cout<<"\n\t enter integers for set2";
               s2.read();
               s3=s1+s2;
              cout<<endl<<"s1=";
               s1.show();
              cout<<endl<<"s2=";
```

```
                s2.show();
               cout<<endl<<"s1 union s2:";
                s3.show();
               cout<<"\n\t enter input characters of set u:";
                s4.read();
               cout<<endl<<"s4=";
                s4.show();
}
```

output:

```
        enter  integers for set1
       input an element:1
              another (Y/N) ?y

       input an element:2
              another (Y/N) ?n

       enter integers for set2
       input an element:2
              another (Y/N) ?y

       input an element:3
              another (Y/N) ?n

s1= 1 2

s2= 2 3

s1 union s2: 1 2 3

       enter input characters of set u:
       input an element:e
              another (Y/N) ?y

       input an element:t
              another (Y/N) ?n

s4= e t
```

10.6 Summary

- Polymorphism means one name having multiple forms.
- Types of polymorphism namely compile time and run time polymorphism are studied.
- Object pointers are useful in creating objects at run time.
- Run time polymorphism is achieved using virtual functions.
- Generic programming using templates by defining function templates or class templates

10.7 Technical Terms

Class template: It is a generic class definition.

Dynamic Binding: The selection (binding) of the function is done dynamically at the runtime. The compiler is able to select the appropriate overloaded member function made in the function call.

Function template: It is a generic function definition.

Instantiation: A process of creating a specific class from a class template.

Template: A feature of C++ that allows generic programming.

Template class: A class create from a class template .

Template function: A specific function created from a function template.

Virtual function: A function qualified by the 'virtual' keyword. When a virtual function is called, the class of the object pointed to determines which function definition has to be used.

10.7 Model Questions:

1. Write notes on virtual functions.
2. What is a Function template? Explain the syntax of function template.

3. What is class template? Explain the syntax of function template.
4. What is algorithm abstraction.

10.8 References

Object-oriented programming with C++,
by E. Bala Gurusamy.

Problem solving with C++
by Walter Savitch

Mastering C++
by K.R.Venugopal, RajkumarBuyya, T.RaviShankar

AUTHOR

M. NIRUPAMA BHAT, MCA., M.Phil.,
Lecturer,
Dept. Of Computer Science,
JKC College,
GUNTUR.

Lesson 11 : Stacks and Queues

Objectives:

In this chapter you will:

- Learn about stacks
- Examine various stack operations
- Learn how to implement a stack as an array
- Learn how to implement a stack as a linked list
- Discover stack applications
- Learn about queues
- Examine various queue operations
- Learn how to implement a queue as an array
- Learn how to implement a queue as a linked list
- Discover queue applications

Structure of the Lesson:

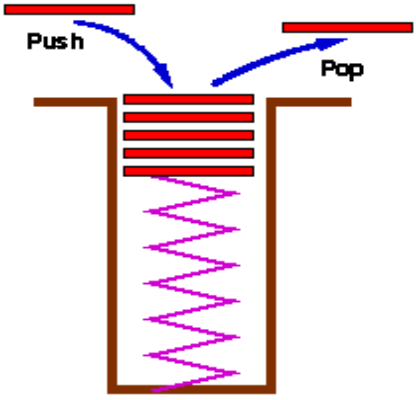
11.1.Introduction

A **data object** is a set of instances or values. The individual instances of a data object are either primitive (or atomic) or composed of instances of another data object. If an individual instance of a data object is not atomic, then its components are called elements. The instances of a data object as well as the elements that constitute individual instances are generally related in some way. In addition to that a set of functions is generally associated with any data object. These functions may transform one instance of an object into another instance of that object, and into an instance of another object also. It may create a new instance without transforming the instances from which the new instance is created.

Boolean, Digit, Letter, NaturalNumber, Integer etc… are examples of data objects. True and False are the instances of Boolean. 0,1,…,9 are the instances of Digit.

A **data structure** is a data object together with the relationships that exist among the instances and among the individual elements that compose an instance.

Stacks and Queues are most frequently used data structures. Both are linear data structures. A stack is called a Last-In First-Out (LIFO) list. Elements are pushed onto the stack at one end, called top of the stack, and removed from the same end. The last element pushed on to the stack is the one to come out first. A common model of a stack is a plate or coin stacker. Plates are "pushed" onto to the top and "popped" off the top.

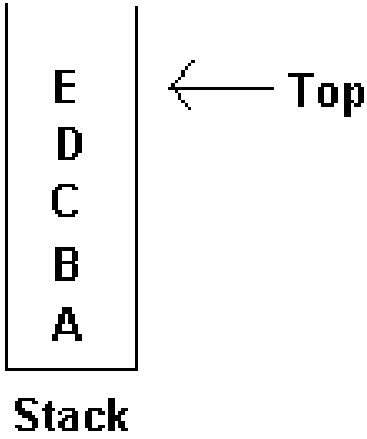


A stack data structure is generally implemented with two principle operations push and pop.

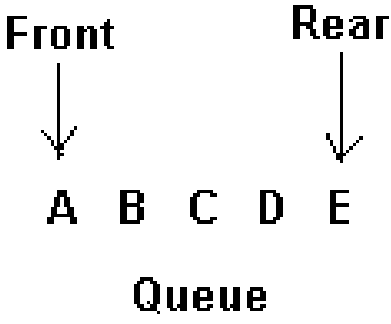
Push : Adds an item to the stack

Pop : Removes the most recently pushed item from the stack.

Given a stack S=(A_1, A_2, A_3, … A_n) then we say that a1 is the bottommost element and element A_i) is on top of element A_{i-1}, 1<i<=n.



The restrictions on a stack imply that if the elements A, B, C, D, E are added to the stack, in that order, then the first element to be removed/deleted must be E. Equivalently we say that the last element to be inserted into the stack will be the first to be removed. For this reason stacks are sometimes referred to as Last In First Out (LIFO) lists.

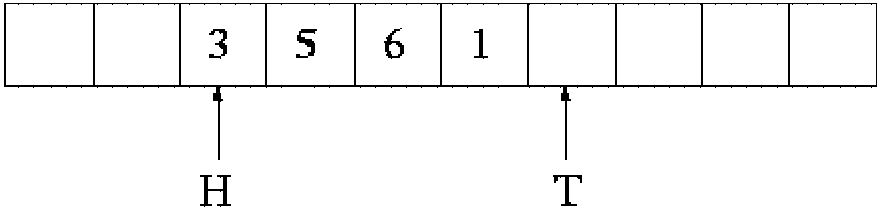


A Queue is First-In First-Out (FIFO) data structure. Elements are inserted into a queue from the 'rear end' and removed from the 'front end'. Given a queue $Q=(A_1, A_2, A_3, \ldots, A_n)$ then we say that A_1 is the first element and the element A_i is in front of the element A_{i-1}, $1<i<=n$.

If the elements A, B, C, D, E are added to the queue, in that order, then the first element to be removed/deleted must be A. Equivalently we say that the first element to be inserted into the queue will be the first to be removed. For this reason queues are referred to as First In First Out (FIFO) lists.

Circular Queue

A circular queue is a particular implementation of a **queue**. It is very efficient and quite useful. A circular queue consists of an array that contains the items in the queue, two array indexes and an optional length. The indexes are called the *head* and *tail* pointers and are labeled H and T on the diagram.



The head pointer points to the first element in the queue, and the tail pointer points just beyond the last element in the queue. If the tail pointer is before the head pointer, the queue wraps around the end of the array.

The problem with circular queue is that, having the head and tail point to the same element would indicate both an empty queue and a full queue. There are two ways around this: either maintain a variable with the number of items in the queue, or create the array with one more element that you will actually need so that the queue is never full.

Insertion and deletion are very simple in a circular queue. To insert, write the element to the tail index and increment the tail, wrapping if necessary (using modulo arithmetic). To delete, save the head element and increment the head, wrapping if necessary (using modulo arithmetic). Instead of using a modulus operator for wrapping (mod in Pascal, % in C) you can use an if statement or (even better) make the size of the array a power of two and simulate the mod with a binary and (& in C).

11.2. Stack Abstract Data Type

As an abstract data type, the stack is a container (data structure) of nodes and has two basic operations: *push* and *pop*. *Push* adds a given node to the top of the stack leaving previous nodes below. *Pop* removes and returns the current top node of the stack. The Stack ADT is given below:

AbstractDataType Stack{
instances
Linear list of elements, one end called top.
operations
Create (): Create an empty stack;
IsEmpty(): Return true if the stack is empty, false otherwise;
IsFull(): Return true if the stack is full, false otherwise;
Top(): Return top element of stack;
Push(x): Place the element x on the top of the stack;

```
        Pop(x) : Remove the top element from stack and
                 assign it to x;
}
```

11.3. Formula based representation of Stack ADT

In formula based representation, a single dimensional array is used to contain the elements of the stack. The following C++ code implements the stack ADT using a linear array.

```
template <class T>
class Stack{
  //LIFO objects.
  public:
      Stack(int maxStacSize=10);
      ~Stack() { delete[] stackList; }
      bool isEmpty() const { return (top == -1); }
      bool isFull() const { return (top == maxSize); }
      T top() const;
      Stack<T>& push(const T& x);
      Stack<T>& pop(T& x);
  private:
      int top, maxSize;
      T  *stackList;
};

template<class T>
Stack<T>::Stack(int maxStackSize)
{// Stack constructor.
      maxSize = maxStackSize – 1;
      stackList = new T[maxStackSize];
      top = -1;
}



template<class T>
```

```
Stack<T>::Top() const
{//Return top element.
        if(isEmpty()) throw OutOfBounds();
        else   return stackList[top];
}

template<class T>
Stack<T>& Stack<T>::push(const T& x)
{// Add x to stack.
        if(isFull()) throw NoMem();
        stackList[++top] = x;
        return *this;
}

template<class T>
Stack<T>& Stack<T>::pop(T& x)
{//pop top element from stack and put it in x.
        if(isEmpty()) throw OutOfBounds();
        x = stackList[top--]; return *this;
 }
```

11.4. Linked representation of Stack ADT

The previous section gave the array representation of stack. Although it is an elegant method, it may lead to wastage of memory space when multiple stacks are to coexist in memory. In such cases stacks can be represented efficiently using a linked list for each stack. The following C++ code gives the linked list implementation of stack.

```
template <class Type>
class Node
{
  public:
        Type info;
```

```
        Node<Type> *link;
};

template<class Type>
class linkedStackType
{
public:
    bool isEmptyStack() const;
      //Function to determine whether the stack is empty.
      //Postcondition: Returns true if the stack is empty,
      //               otherwise returns false.

    void destroyStack();
      //Function to remove all the elements of the stack,
      //leaving the stack in an empty state.
      //Postcondition: stackTop = NULL

    void push(const Type& newItem);
      //Function to add newItem to the stack.
      //Precondition: The stack exists and is not full.
      //Postcondition: The stack is changed and newItem
      //                    is added to the top of the stack.

    Type top() const;
      //Function to return the top element of the stack.
      //Precondition: The stack exists and is not empty.
      //Postcondition: If the stack is empty, the program
      //               terminates; otherwise, the top element
      //               of the stack is returned.

    void pop(Type& item);
      //Function to remove the top element of the stack.
      //Precondition: The stack exists and is not empty.
      //Postcondition: The stack is changed and the top
      //element is removed from the stack.



    linkedStackType();
```

```
    //default constructor
    //Postcondition: stackTop = NULL

private:
   Node<Type> *stackTop; //pointer to the stack

};


template<class Type> //default constructor
linkedStackType<Type>::linkedStackType()
{
      stackTop = NULL;
}

template<class Type>
void linkedStackType<Type>::destroyStack()
{
      Node<Type> *temp; //pointer to delete the node

      while (stackTop != NULL)
            //while there are elements in the stack
      {
        temp = stackTop;
          //set temp to point to the current node
        stackTop = stackTop->link;
         //advance stackTop to the next node
        delete temp;
         //deallocate memory occupied by temp
      }
}// end destroyStack

template<class Type>
bool linkedStackType<Type>::isEmptyStack() const
{
      return(stackTop == NULL);
}
```

```
template<class Type>
void linkedStackType<Type>::push(const Type&
newElement)
{
  Node<Type> *newNode;
       //pointer to create the new node
  newNode = new Node<Type>; //create the node
  assert(newNode != NULL);
  newNode->info = newElement;
      //store newElement in the node
  newNode->link = stackTop;
      //insert newNode before stackTop
  stackTop = newNode;
      //set stackTop to point to the top node
} //end push


template<class Type>
Type linkedStackType<Type>::top() const
{
      assert(stackTop != NULL);
         //if stack is empty terminate the program
      return stackTop->info;     //return the top element
}//end top

template<class Type>
void linkedStackType<Type>::pop(Type& item)
{
  Node<Type> *temp;
    //pointer to deallocate memory
  if (stackTop != NULL)   {
      temp = stackTop;
        //set temp to point to the top node
      stackTop = stackTop->link;
       //advance stackTop to the next node
      item = temp->info;
      delete temp;   //delete the top node
  }
```

```
 else
        cout << "Cannot remove from an empty stack."
             << endl;
}//end pop


    //destructor
template<class Type>
linkedStackType<Type>::~linkedStackType()
{
        destroyStack();
}//end destructor
```

11.5. Stack Applications

Most of the high-level language programs (even C!) make use of a stack frame for the working memory of each procedure or function invocation. When any procedure or function is called, a number of words (together called a *stack frame*) are pushed onto a program stack. When the procedure or function returns, this frame of data is popped off the stack. Like this there are many applications of stack data structure in computer science.

Towers of Hanoi problem – A stack application.

In the Towers of Hanoi problem, you are given n disks and three towers. The disks are initially stacked on tower 1 in increasing order of size from bottom to top. You are to move the disks to tower 2 using tower 3, one disk at a time, such that no disk is ever on top of a smaller one. A solution for this problem using recursive method is given below:

step 1 : move the top n-1 disks from tower 1 to tower 3.

step 2 : move the bottom disk from tower 1 to tower 2.

step 3 : move the n-1 disks from tower 3 to tower 2.

The following is a recursive function to solve Towers of Hanoi problem.

```
void towersOfHanoi(int n, int x, int y, int z)
{
  // Move the top n disks from tower x to tower y.
  // Use tower z fro intermediate storage.
        if(n>0)
        {
          towersOfHanoi(n-1, x, z, y);
          cout<<"move top disk from tower "<<x
               <<" to top of tower "<<y<<endl;
          towersOfHanoi(n-1, z, y, x);
         }
}
```

A non-recursive function can be written for this problem using a formula based stack. The following C++ code gives a non-recursive function to solve the Towers of Hanoi problem.

```
class Hanoi{
   friend void TowersOfHanoi(int);
   public:
        void TowersOfHanoi(int n, int x, int y, int z);
   private:
        Stack<int> *s[4]; // array of pointers to stacks.
};

void Hanoi::TowersOfHanoi(int n, int x, int y, int z)
```

```
{//Move the top n disks from tower x to tower y.
  //Use tower z for intermediate storage.
   int d;  // disk nuber.
   if(n > 0) {
        TowersOfHanoi(n-1, x, z, y);
        s[x]->pop(d);
        s[y]->push(d);
        showstate();
        TowersOfHanoi(n-1, z, y, x);
   }
}


void TowersOfHanoi(int n)
{
   // Preprocessor for Hanoi::towersOfHanoi.
   Hanoi x;
   // create three stacks of size n each.
   x.s[1] = new Stack<int> (n);
   x.s[2] = new Stack<int> (n);
   x.s[3] = new Stack<int> (n);

   for(int d = n; d > 0; d--) // initialize
        x.s[1] ->push(d);  // add disk d to tower 1.
   // move n disks from tower 1 to tower 3
   //using 2 as intermediate.
   x.TowersOfHanoi(n, 1, 2, 3);
}
```

11.6. Queue Abstract Data Type

Queue is a First-In First-Out data structure. Elements are inserted into the queue at the rear end. They are removed from the queue at the front end. The Queue data structure is implemented with two basic operations insert() and delete(). Corresponding to the two **ends** this data structure uses two index variables or pointer variables. The Queue ADT is given below:

AbstractDataType Queue {

instances

ordered list of elements. One end is called *front*, the other *rear*.

operations:

Create(): Create an empty queue;
IsEmpty(): Return true if queue is empty, return false otherwise;
IsFull(): Return true if queue is full, false otherwise;
First() : Return first element of queue;
Last() : Return last element of queue;
Insert(x) : Add element x to the queue;
Delete(x) : Delete front element from queue and put it in x;

}

11.7. Formula Based Representation of Queue ADT

The formula based representation uses an linear array to hold the queue elements, two index variables front and rear and an optional count variable. The following C++ code shows the formula based representation of queue.

```
template<class Type>
class Queue
{
public:

    bool isEmpty () const;
      //Function to determine whether the queue is empty.
      //Postcondition: Returns true if the queue is empty,
      //               otherwise returns false.

    void initializeQueue();
      //Function to initialize the queue to an empty state.
      //Postcondition: count = 0; queueFront = 0;
      //               queueRear = maxQueueSize – 1

   bool isFull () const;
   void destroy ();
      //Function to remove all the elements from the queue.
      //Postcondition: count = 0; queueFront = 0;
      //               queueRear = maxQueueSize – 1

    Type first() const;
      //Function to return the first element of the queue.
      //Precondition: The queue exists and is not empty.
      //Postcondition: If the queue is empty, the program
      //               terminates; otherwise, the first
      //               element of the queue is returned.
    Type last() const;
      //Function to return the last element of the queue.
      //Precondition: The queue exists and is not empty.
      //Postcondition: If the queue is empty, the program
      //               terminates; otherwise, the last
      //               element of the queue is returned.

    void insert(const Type& queueElement);
      //Function to add queueElement to the queue.
      //Precondition: The queue exists and is not full.
      //Postcondition: The queue is changed and
            queueElement is added to the queue.
```

```
    void deleteQ ();
      //Function to remove the first element of the queue.
      //Precondition: The queue exists and is not empty.
      //Postcondition: The queue is changed and the first
      //               element is removed from the queue.

    Queue(int queueSize = 100);
      //constructor
    ~Queue();
      //destructor

private:
    int maxQueueSize; //variable to store the maximum
                      //queue size
    int count;        //variable to store the number of
                      //elements in the queue
    int queueFront;   //variable to point to the first
                      //element of the queue
    int queueRear;    //variable to point to the last
                      //element of the queue
    Type *list;       //pointer to the array that holds
                      //the queue elements
};

template<class Type>
void Queue<Type>::initializeQueue()
{
        queueFront = 0;
    queueRear = maxQueueSize - 1;
        count = 0;
}
template<class Type>
void Queue<Type>::destroy ()
{
        queueFront = 0;
    queueRear = maxQueueSize - 1;
        count = 0;
}
```

```
template<class Type>
bool queueType<Type>::isFull () const
{
  return(count == maxQueueSize);
}

template<class Type>
bool Queue<Type>::isEmpty () const
{
  return(count == 0);
}

template<class Type>
void Queue<Type>::insert(const Type& newElement)
{
  if (!isFull ())   {
       queueRear = (queueRear + 1) % maxQueueSize;
      //use mod operator to advance queueRear because
     //the array is circular
       count++;
       list[queueRear] = newElement;
  }
  else   cout << "Cannot add to a full queue." << endl;
}

template<class Type>
Type Queue<Type>::first() const
{
  assert(!isEmpty());
  return list[queueFront];
}

template<class Type>
Type Queue<Type>::last() const
{
    assert(!isEmpty());
    return list[queueRear];
}
```

```
template<class Type>
void Queue<Type>::deleteQ()
{
  if (!isEmpty())
  {
      count--;
      queueFront = (queueFront + 1) % maxQueueSize;
            //use the mod
            //operator to advance queueFront
            //because the array is circular
  }
  else
      cout << "Cannot remove from an empty queue" <<
endl;
}

//constructor
template<class Type>
Queue<Type>::Queue(int queueSize)
{
   if (queueSize <= 0)    {
      cout << "Size of the array to hold the queue must "
            << "be positive." << endl;
      cout << "Creating an array of size 100." << endl;

      maxQueueSize = 100;
   }
   else
      maxQueueSize = queueSize;
            //set maxQueueSize to queueSize

   queueFront = 0;  //initialize queueFront
   queueRear = maxQueueSize - 1;  //initialize queueRear
   count = 0;
   list = new Type[maxQueueSize];
            //create the array to
            //hold the queue elements
      assert(list != NULL);
}
```

```
template<class Type>
Queue<Type>::~Queue()   //destructor
{
  delete [] list;
}
```

11.8. Linked representation of queue ADT

The linked representation uses a linear linked list to hold the queue elements. Pointers to the first and last nodes in the list are stored in pointer variables front and rear. New nodes are inserted using rear pointer. Nodes are deleted from the queue using front pointer.

```
template <class Type>
class QNode
{
public:
       Type info;
       QNode<Type> *link;
};

template<class Type>
class linkedQueueType
{
public:

    bool isEmpty() const;
      //Function to determine whether the queue is empty.
      //Postcondition: Returns true if the queue is empty,
      //               otherwise returns false.
    bool isFull() const;
      //Function to determine whether the queue is full.
      //Postcondition: Returns true if the queue is full,
      //               otherwise returns false.
```

```
void destroy();
  //Function to delete all the elements from the queue.
  //Postcondition: queueFront = NULL;
  //queueRear = NULL
void initializeQueue();
  //Function to initialize the queue to an empty state.
  //Postcondition: queueFront = NULL;
  //queueRear = NULL

Type first() const;
  //Function to return the first element of the queue.
  //Precondition: The queue exists and is not empty.
  //Postcondition: If the queue is empty, the program
  //               terminates; otherwise, the first
  //               element of the queue is returned.
Type last() const;
  //Function to return the last element of the queue.
  //Precondition: The queue exists and is not empty.
  //Postcondition: If the queue is empty, the program
  //               terminates; otherwise, the last
  //               element of the queue is returned.

void insert(const Type& queueElement);
  //Function to add queueElement to the queue.
  //Precondition: The queue exists and is not full.
  //Postcondition: The queue is changed and
  // queueElement is added to the queue.

void deleteQueue();
  //Function  to remove the first element of the queue.
  //Precondition: The queue exists and is not empty.
  //Postcondition: The queue is changed and the first
  // element is removed from the queue.

linkedQueueType();
  //default constructor
~linkedQueueType(); //destructor
```

```
private:
   QNode<Type> *queueFront; //pointer to the front of
                            //the queue
   QNode<Type> *queueRear;  //pointer to the rear of
                            //the queue
};

template<class Type>
linkedQueueType<Type>::linkedQueueType()      //default
constructor
{
      queueFront = NULL; // set front to null
      queueRear = NULL;  // set rear to null
}

template<class Type>
bool linkedQueueType<Type>::isEmpty() const
{
      return(queueFront == NULL);
}
template<class Type>
bool linkedQueueType<Type>::isFull() const{
      return false;
}
template<class Type>
void linkedQueueType<Type>::destroy()
{
      QNode<Type> *temp;
      while(queueFront!= NULL){
       //while there are elements left in the queue
         temp = queueFront;
             //set temp to point to the current node
         queueFront = queueFront->link;
       //advance first to the next node
         delete temp;
             //deallocate memory occupied by temp
}
   queueRear = NULL;  //set rear to NULL
}
```

```
template<class Type>
void linkedQueueType<Type>::initializeQueue()
{
    destroyQueue();
}

template<class Type>
void       linkedQueueType<Type>::insert(const       Type&
newElement)
{
       QNode<Type> *newNode;

       newNode = new QNode<Type>;  //create the node
       assert(newNode != NULL);

       newNode->info = newElement;    //store the info
       newNode->link = NULL;
           //initialize the link field to NULL
       if(queueFront == NULL)
          //if initially the queue is empty
       {
          queueFront = newNode;
          queueRear = newNode;
       }
       else   //add newNode at the end
       {
          queueRear->link = newNode;
          queueRear = queueRear->link;
       }
}//end insert

template<class Type>
Type linkedQueueType<Type>::first() const
{
       assert(queueFront != NULL);
       return queueFront->info;
}
```

```
template<class Type>
Type linkedQueueType<Type>::last() const
{
	assert(queueRear!= NULL);
	return queueRear->info;
}

template<class Type>
void linkedQueueType<Type>::deleteQueue()
{
  QNode<Type> *temp;

  if(!isEmpty())
  {
	temp = queueFront;
		//make temp point to the first node
	queueFront = queueFront->link;
		 //advance queueFront
	delete temp;
		//delete the first node
	if(queueFront == NULL)
		//if after deletion the queue is empty
	queueRear = NULL;
	     //set queueRear to NULL
  }
  else
	cerr<<"Cannot remove from an empty queue “
	    <<endl;
}//end deleteQueue

template<class Type>
linkedQueueType<Type>::~linkedQueueType()
//destructor
{
	QNode<Type> *temp;
	while(queueFront != NULL)
	 //while there are elements left in the queue
	{
	  temp = queueFront;
```

```
            //set temp to point to the current node
        queueFront = queueFront->link;
            //advance first to the next node
        delete temp;
      //deallocate memory occupied by temp
      }
      queueRear = NULL;  // set rear to null
}
```

11.9. Queue Applications

Railroad Car Rearrangement Problem

Problem description: A freight train has n railroad cars. Each is to be left at a different station. Assume that the n stations are numbered 1 through n and that the freight train visits these stations in the order n through 1. The railroad cars are labeled by their destination. To facilitate removal of the railroad cars from the train, we must reorder the cars so that they are in the order 1 through n from front to back. When the cars are in this order, the last car is detached at each station. The cars are rearranged at a shunting yard that has an input track, an output track, and k holding tracks between the input and output tracks. The following figure shows a shunting yard with k=3 holding tracks H1, H2, and H3. The n cars of the freight train begin in the input track and are to end up in the output track in the order 1 through n from right to left.

To rearrange the cars, we examine the cars on the input track from front to back. If the car being examined is the next one in the output arrangement, we move it directly to the output track. If not, we move it to a holding track and leave it there until it is time to place it in the output. The holding tracks operate in a FIFO manner as cars enter and leave these tracks from the top. When rearranging cars only the following moves are allowed.

1. A car may be moved from the front of the input track into one of the holding tracks or the left of the output track.
2. A car may be moved from the front of a holding track to the left end of the output track.

Solution

When a car is to be moved to a holding track, we can use the following track selection to decide which holding track to move it to. " Move car '**C'** to a holding track that contains only cars with a smaller label; if there are several such tracks, select one with largest label at its left end; otherwise, select an empty track (if one remains).

Program to rearrange cars using queues.

```
void Output (int& minH, int& minQ,
                LinkedQueue<int> H[], int k, int n)
{
  // Move from hold to output and update minH and minQ.
  int c; // car index.
  // delete smallest car minH from queue minQ.
  H[minQ].deleteQ(c);
  cout<< "Move car "<<minH<< " from holding track"
       << minQ << " to output "<< endl;
// find new minH and minQ by checking front of all queues.
  MinH = n + 2;
for( int i=1; i<= k; i++)
  if( !H[I].IsEmpty() && (c = H[I].First()) < minH)
  {
      minH = c;
      minQ = i;
  }
}

bool Hold( int c, int& minH, int &minQ,
            LinkedQueue<int> H[], int k)
{// Add car C to a holding track.
 // Return false if no feasible holding track.
 // Return true otherwise.
```

```
  // find best holding track for car c initialize.

   int BestTrack = 0, BestLast = 0, x;

// scan holding tracks.
for( int i=1; i <= k; i++)
   if(!H[i].IsEmpty())
   {
        x = H[i].Last();
        if( c > x && x > BestLast)
        {
               BestLast = x;
               BestTrack = i;
        }
   }
   else if (!BestTrack) BestTrack = i;
   if (!BestTrack) return false;
        H[BestTrack].insert(c);
   Cout<<" Move car "<< c << " from input "
         << " to holding track "<< BestTrack << endl;
   if(c< minH)
   {
     minH = c;
     minQ = BestTrack;
   }
   return true;
}
```

11.10. Summary

A stack is simply another collection of data items and thus it would be possible to use exactly the same specification as the one used for our general collection. However, collections with the LIFO semantics of stacks are so important in computer science that it is appropriate to set up a limited specification appropriate to stacks only.

Although a linked list implementation of a stack is possible (adding and deleting from the head of a linked list produces exactly the LIFO semantics of a stack), the most common applications for stacks have a space restraint so that using an array implementation is a natural and efficient one.

Like stacks, queues can be used to remember the search space that needs to be explored at one point of time in traversing algorithms. *Breadth-First* search of a graph uses a queue to remember the nodes yet to be visited.

A queue is natural data structure for a system to serve the incoming requests. Most of the process scheduling or disk scheduling algorithms in operating systems use queues. Computer hardware like a processor or a network card also maintain buffers in the form of queues for incoming resource requests. A stack like data structure causes starvation of the first requests, and is not applicable in such cases. A mailbox or port to save messages to communicate between two users or processes in a system is essentially a queue like structure.

11.11. Technical Terms:

Pop
This operation removes the top element of the stack and stores the top element into a location called poppedElement

Push
This operation places a new element on top of the stack

Program stack
A data structure, which the computer uses to implement function calls among numerous other applications

Queue
A data structure in which the elements are added at one end, called the rear, and deleted from the other end

Stack
A data structure in which the elements are added and removed from one end only

Circular Queue
An implementation of queue data structure, in which the first and last positions, in the container array are treated as adjacent.

11.12. Model Questions:

1. Write a C++ program to test the array implementation of stack.
2. Write about different types of queues.
3. What is a double-ended queue?
4. Explain how stack is useful in solving towers of Hanoi?
5. Write a C++ program for linked representation of queue.

11.13. References:

Data Structures, Algorithms, and Applications in C++ by SAHNI.

AUTHOR:

Y. VENKATESWARA RAO, M.C.A.,
Lecturer,
Dept.Of Computer Science,
JKC College,
GUNTUR

Lesson 12: Linked Lists

Objectives:

To:

- Learn about linked lists
- Become aware of the basic properties of linked lists
- Explore the insertion and deletion operations on linked lists
- Discover how to build and manipulate a linked list
- Learn how to construct a doubly linked list

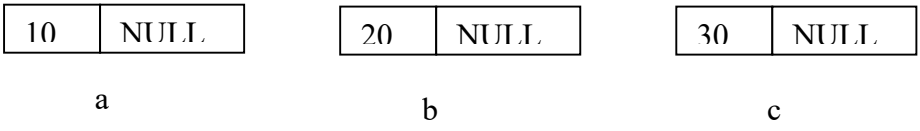
Structure of the Lesson:

12.1.Introduction

A *linked list* is a chain of *structs* or records called nodes. Each node has at least two members, one of which points to the next item or node in the list. These are defined as *Single Linked Lists* because they only point to the next item, and not the previous. Those that do point to both are called *Doubly Linked Lists* or *Circular Linked Lists*. According to this definition, the linked list record can hold **any data.** The only drawback is that each record must be an instance of the same structure. This means that we couldn't have a record with a char pointing to another structure holding a short, a char array, and a long. Another aspect of the linked list is that, each structure can be located anywhere in memory each node doesn't have to be linear in memory.

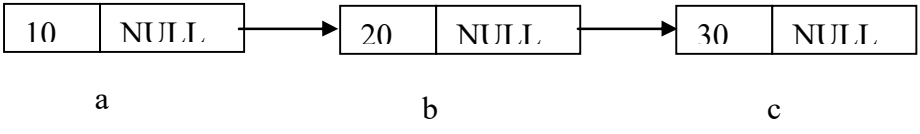
Single Linked List:

There are several kinds of linked data structures. One among them is the linear linked list. In this data structure all the nodes in the list are linked linearly in sequential order. The nodes in linear linked list before they are linked, appear as shown below.



The above three nodes can be linked to form a linear linked list.

After linking the nodes a, b ,c the list looks as:



Linked List Representation:

One major advantage of a linear linked list is that it can be of varying length. The number of nodes in the list can be dynamically decided, as each node can be dynamically allotted. The nodes of a linked list need not be components of an array. No formula is used to locate individual nodes. Each node keeps explicit information about the location of other relevant nodes. This information is called link(s) or pointer(s). In a single linked list each node contains a link to its succeeding node. In a double linked list each node contains links to its succeeding as well as preceding nodes. In a circular linked list the last node of the list contains a pointer to the first node in the list.

The nodes of a linked list are dynamically created using **new** operator in C++. The new operator is called dynamic memory allocation operator. It allocates the required memory for a node from the heap area. If it fails due to insufficient memory, it returns a NULL value. These nodes are created using a template or blueprint defined using either **struct** or **class** data construct. This template is a self-referential structure, i.e. it contains one or more pointer variables of its own type. These pointer variables are filled with the memory addresses of other nodes in the list, so they work as links.

12.2.Linear Linked List ADT.

A linear linked list may be specified as an abstract data type (ADT) in which we provide a specification of the instances as well as of the operations that are to be performed. The abstract data type specification is independent of any representation. All representations of the abstract data type must satisfy the specification. So

these representations can be used interchangeably in the applications of this data structure.

Abstract Data Type *LinearList* {
  Instances
    Ordered finite collections of zero or more elements.
  Operations
    Create(): create an empty linear list
    Destroy(): erase the list
    IsEmpty(): return *true* if the list is empty, *false* otherwise.
    Length(): return the list size (i.e., the number of elements in the list.)
    Find(K,x): return the k^{th} element of the list in x
      return *false* if there is no k^{th} element.
    Search(x): return the position of x in the list
      return 0 if x is not in the list.
    Insert(k,x): insert x just after the k^{th} element
      function returns the modified linear list.
    Output(out): put the list into the output stream out;
}

12.3. Linear Linked List ADT Implementation

```
template <class Type>
class LLNode
{
  public:
      Type info;
      LLNode<Type> *link;
};

template<class Type>
class LinearLinkedList
{
```

```
public:
    void create();
      //Initialize the list to an empty state.
      //Postcondition: first = NULL, last = NULL, count = 0;

    bool isEmptyList() const;
      //Function to determine whether the list is empty.
      //Postcondition: Returns true if the list is empty,
      //               otherwise it returns false.

    void print() const;
      //Function to output the data contained in each node.
      //Postcondition: none

    int length() const;
      //Function to return the number of nodes in the list.
      //Postcondition: The value of count is returned.

    void destroyList();
      //Function to delete all the nodes from the list.
      //Postcondition: first = NULL, last = NULL, count = 0;

    bool find( int k, Type& x);
      //Function to find the k^th element of the list.
      //Precondition: The list and the k^th element must exist.
      //Postcondition: If the list is empty, the program
      //          returns false.
      //          k^th element of the list is returned through x.
bool search(const Type& searchItem) const;
  //Function to determine whether searchItem is in the list.
      //Postcondition: Returns true if searchItem is in the
      // list, otherwise the value false is  returned.

 void insertFirst(const Type& newItem);
   //Function to insert newItem at the beginning of the list.
      //Postcondition: first points to the new list, newItem is
      // inserted at the beginning of the list, last points to
   //the last node in the list, and count is incremented by 1.
```

```
    void insertLast(const Type& newItem);
      //Function to insert newItem at the end of the list.
      //Postcondition: first points to the new list, newItem
      //               is inserted at the end of the list,
      //               last points to the last node in the list,
      //               and count is incremented by 1.

    void deleteNode(const Type& deleteItem);
      //Function to delete deleteItem from the list.
      //Postcondition: If found, the node containing
      //               deleteItem is deleted from the list.
      //               first points to the first node, last
      //               points to the last node of the updated
      //               list, and count is decremented by 1.


    LinearLinkedList ();
      //default constructor
      //Initializes the list to an empty state.
      //Postcondition: first = NULL, last = NULL, count = 0;

    ~ LinearLinkedList ();
      //destructor
      //Deletes all the nodes from the list.
      //Postcondition: The list object is destroyed.

protected:
    int count;
    //variable to store the number of nodes in the list
    LLNode<Type> *first;
      //pointer to the first node of the list
    LLNode<Type> *last;
       //pointer to the last node of the list
};
```

12.4. Basic list operations

The basic operations on a linear linked list, as stated in the ADT, are implemented as member functions of its class. These operations are as follows:

1. creating a list.
2. inserting an element into the list.
3. searching for an element.
4. finding whether the list is empty..
5. finding the length of a list.
6. deleting an element from the list.

The first operation on a linear linked list is to create it. The following function creates an empty list, and assigns NULL to the pointers first and last.

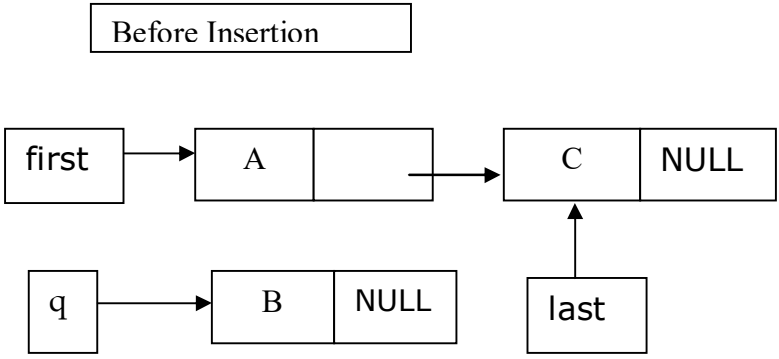
```
template<class Type>
void LinearLinkedList <Type>::create()
{
	first = NULL;
	last = NULL;
	count = 0;
}

template<class Type>
LinearLinkedList <Type>::LinearLinkedList () //default constructor
{
	first = NULL;
	last = NULL;
	count = 0;
}
```

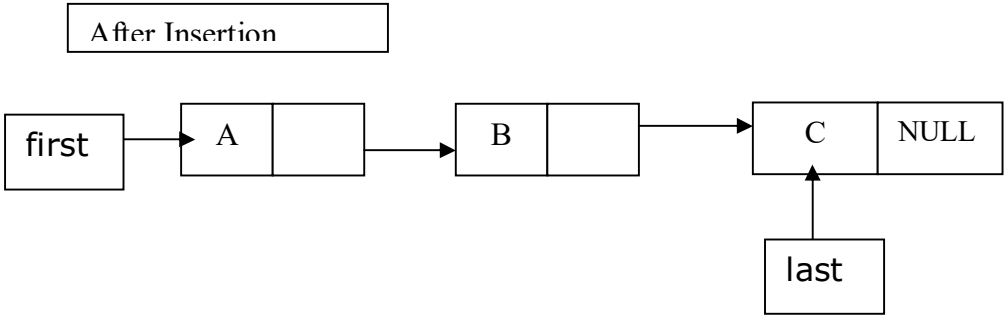
Elements can be inserted into a linear linked list at any position. However it is easy to insert nodes at the beginning and at the end. The following two member functions insert new nodes at the front and at the last. When a new node is inserted at the front of the list the first variable is changed to point to the new node. When the new node is inserted at the last, the last variable is changed to point to the newly inserted node.

```
template<class Type>
void LinearLinkedList <Type>::insertFirst(const Type&
newItem)
{
   LLNode<Type> *newNode;
    //pointer to create the new node
   newNode = new LLNode<Type>; //create the new node
   assert(newNode != NULL);
    //if unable to allocate memory terminate the program

   newNode->info = newItem;
     //store the new item in the node
   newNode->link = first;
     //insert newNode before first
   first = newNode;
    //make first point to the actual first node
   count++;
 //increment count

   if (last == NULL)
/*if the list was empty, newNode is alsothe last node in the
list*/
    last = newNode;
}//end insertFirst

template<class Type>
void LinearLinkedList <Type>::insertLast(const Type&
newItem)
{
   LLNode<Type> *newNode;
      //pointer to create the new node
   newNode = new LLNode<Type>;
      //create the new node
   assert(newNode != NULL);
  //if unable to allocate memory,terminate the program
   newNode->info = newItem;
       //store the new item in the node
   newNode->link = NULL;
       //set the link field of newNode to NULL
```

```
  if (first == NULL)  //if the list is empty, newNode is
                      //both the first and last node
  {
        first = newNode;
        last = newNode;
        count++;        //increment count
  }
else    //the list is not empty, insert newNode after last
  {
      last->link = newNode; //insert newNode after last
      last = newNode;
       //make last point to the actual last node
    count++;            //increment count
  }
}//end insertLast
```

A new node can be inserted in the middle of a list by giving a specific position to it. To insert the new node as K^{th} node in a list, you should assign the new node address to the $(K-1)^{th}$ node's **next** variable, and the **next** variable of new node gets the $(K+1)^{th}$ node's address.





The linear linked list supports a search operation on it. Using this member function you can search for an element in the list. If the element is found the list returns true, otherwise it returns false.

```
template<class Type>
bool LinearLinkedList <Type>::search(const Type&
searchItem) const
{
   LLNode<Type> *current; //pointer to traverse the list
   bool found = false;

   current = first;
   //set current to point to the first node in the list

   while (current != NULL && !found)    //search the list
      if (current->info == searchItem)
         //searchItem is found
         found = true;
      else
         current = current->link;
     //make current point to the next node
   return found;
}//end search
```

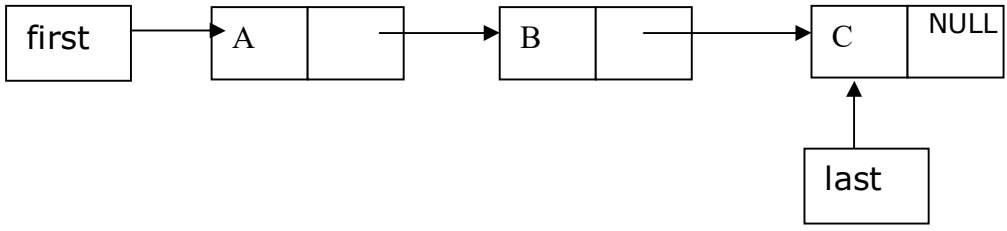
The linearlinkedlist ADT includes a function to check whether it is empty. The isEmptyList() function returns true if the list is empty, otherwise it returns flase. In an empty list the first and last variables contain NULL value.

```
template<class Type>

bool LinearLinkedList <Type>::isEmptyList() const
{
      return(first == NULL);
}
```

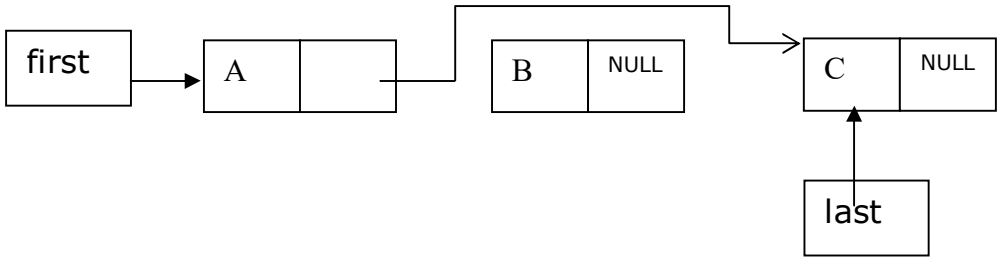
Deleting a node from a linear list:

To delete a node from a list, its previous node is adjusted such that its 'next' member is assigned the successor to the deleted element. If the first node is deleted from the list, the variable **first** is made to point to the second node. Similarly when the last node is deleted from the list the variable **last** made to point to the node before the deleted node. When all the nodes are deleted from the list both the variables **first** and **last** will be assigned NULL.

Before deleting node B



After deleting node B



```
template<class Type>
void LinearLinkedList <Type>::deleteNode(const Type&
deleteItem)
{
   LLNode<Type> *current; //pointer to traverse the list
   LLNode<Type> *trailCurrent;
     //pointer just before current
   bool found;

   if (first == NULL)    //Case 1; the list is empty.
      cout << "Cannot delete from an empty list."
           << endl;
   else
   {
      if (first->info == deleteItem) //Case 2
      {
         current = first;
         first = first->link;
         count--;
         if (first == NULL)    //the list has only one node
            last = NULL;
         delete current;
      }

      else //search the list for the node with the given info
      {
         found = false;
         trailCurrent = first;
           //set trailCurrent to point to the first node
         current = first->link;
          //set current to point to the second node

         while (current != NULL && !found)
         {
            if (current->info != deleteItem)
            {
               trailCurrent = current;
               current = current-> link;
            }
```

```
            else
                found = true;
          }//end while
         if (found) //Case 3; if found, delete the node
          {
            trailCurrent->link = current->link;
            count--;

            if (last == current)
             //node to be deleted was the last node
                last = trailCurrent;
            //update the value of last
            delete current;  //delete the node from the list
          }
          else
            cout << "The item to be deleted is not in "
                 << "the list." << endl;
       }//end else
   }//end else
}//end deleteNode

template<class Type>
LinearLinkedList <Type>::~LinearLinkedList ()
//destructor
{
  destroyList();
}//end destructor

template<class Type>
void LinearLinkedList <Type>::destroyList()
{
   LLNode<Type> *temp;
  //pointer to deallocate the memory occupied by the node
   while (first != NULL)   //while there are nodes in the list
   {
      temp = first;        //set temp to the current node
      first = first->link; //advance first to the next node
      delete temp;
      //deallocate the memory occupied by temp
```

```
    }
    last = NULL; //initialize last to NULL; first has already
         //been set to NULL by the while loop
    count = 0;
}
```

The print function of the linked list class displays all the nodes in the list.

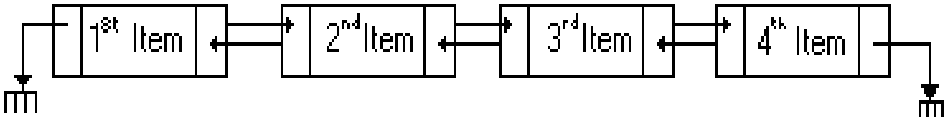
```
template<class Type>
void LinearLinkedList <Type>::print() const
{
       LLNode<Type> *current;
       //pointer to traverse the list

       current = first;
       //set current so that it points to the first node
       while (current != NULL) //while more data to print
       {
          cout << current->info << " ";
          current = current->link;
       }
}//end print
```

The number of nodes in a list can be counted using the length() function. This function returns the count value in the linked list object.

```
template<class Type>
int LinearLinkedList <Type>::length() const
{
   return count;
}  //end length
```

12.5. Doubly Linked List

A doubly linked list is an ordered sequence of nodes in which each node has two pointers: left and right. The left pointer points to the node(if any) on the left, and the right pointer points to the node(if any) on the right. The following figure shows a doubly linked list.



Doubly linked list class can be implemented using the ADT for linear linked list. The following C++ code can be used to create and manipulate a doubly linked list.

```
//Definition of the node
template<class Type>
class DLNode
{
public:
    Type info;
    DLNode<Type> *next;
    DLNode<Type> *back;
};

template<class Type>
class DoublyLinkedList
{
public:

    void initializeList();
      //Function to initialize the list to an empty state.
      //Postcondition: first = NULL; last = NULL; count = 0;

    bool isEmpty() const;
```

```
    //Function to determine whether the list is empty.
    //Postcondition: Returns true if the list is empty,
    //               otherwise returns false.

  void destroy();
    //Function to delete all the nodes from the list.
    //Postcondition: first = NULL; last = NULL; count = 0;

  void print() const;
    //Function to output the info contained in each node.

  void PrintBack() const;
    //Function to output the info contained in each node
    //in reverse order.

  int length()const;
    //Function to return the number of nodes in the list.
    //Postcondition: The value of count is returned.


  bool search(const Type& searchItem) const;
 //Function to determine whether searchItem is in the list.
    //Postcondition: Returns true if searchItem is found in
    //               the list, otherwise returns false.

  void insertNode(const Type& insertItem);
    //Function to insert newItem in the list.
    //Precondition: If the list is nonempty, it must be in
    //              order.
 //Postcondition: insertItem is inserted at the proper place
    //               in the list, first points to the first
    //               node, last points to the last node of the
    //               new list, and count is incremented by 1.

void deleteNode(const Type& deleteItem);
    //Function to delete deleteItem from the list.
  //Postcondition: If found, the node containing deleteItem
    //               is deleted from the list; first points
    //               to the first node of the new list, last
```

```
    //            points to the last node of the new list,
    //            and count is decremented by 1; otherwise,
    //            an appropriate message is printed.

   DoublyLinkedList();
    //default constructor
    //Initializes the list to an empty state.
    //Postcondition: first = NULL; last = NULL; count = 0;

protected:
   int count;
   DLNode<Type> *first; //pointer to the first node
   DLNode<Type> *last;  //pointer to the last node

};
```

A doubly linked list object can be created using the default constructor of the DoublyLinkedList class. The created object will contain an empty list where the first and last variables contain NULL values, and count is assigned zero.

```
template<class Type>
DoublyLinkedList<Type>::DoublyLinkedList()
{
      first= NULL;
      last = NULL;
      count = 0;
}
```

The isEmpty() function returns true if the list is empty, otherwise it returns false. It compares the first variable with NULL and returns the result.

```
template<class Type>
bool DoublyLinkedList<Type>::isEmpty() const
{
   return(first == NULL);
}
```

The list can be entirely removed from memory using the destroy() function. This function removes all the nodes of the list and returns the memory occupied by them to the free list of the operating system. It resets the count to zero. The first and last variables are set to NULL by the end of the function. The initialize() function is to reset the list to initial state of empty list. This function uses the destroy() function to remove all the nodes, then the list becomes empty.

```
template<class Type>
void DoublyLinkedList<Type>::destroy()
{
	DLNode<Type>  *temp; //pointer to delete the node

	while (first != NULL)
	{
		temp = first;
		first = first->next;
		delete temp;
	}

	last = NULL;
	count = 0;
}

template<class Type>
void DoublyLinkedList<Type>::initializeList()
{
	destroy();
}
```

The length() function returns count of nodes in the list. It simply returns the count variable value. For an empty list the this function returns zero.

```
template<class Type>
int DoublyLinkedList<Type>::length() const
{
      return count;
}
```

The print() and printBack() functions display the list in forward and backward directions. The print() function displays the nodes from first node to the last node. The printBack() function display the nodes in the reverse order, i.e., starting from the last to first node.

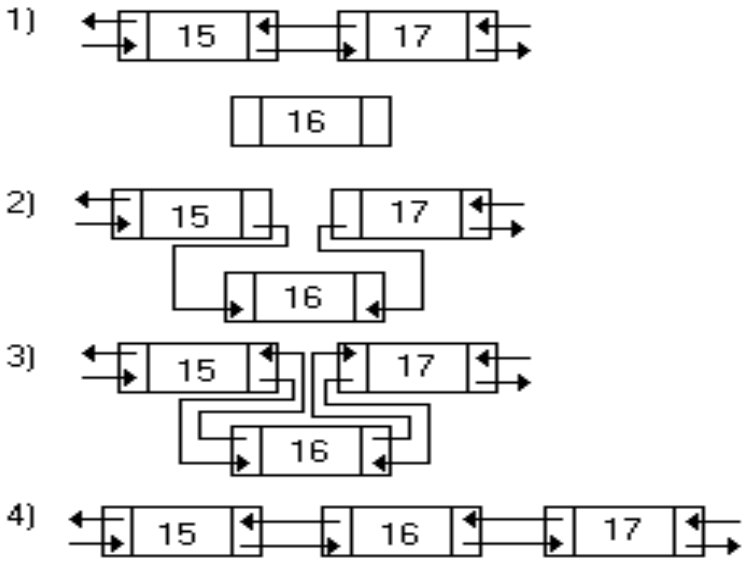
```
template<class Type>
void DoublyLinkedList<Type>::print() const
{
   DLNode<Type> *current;
    //pointer to traverse the list
   current = first;  //set current to point to the first node
   while (current != NULL)
       {
         cout << current->info << "  ";  //output info
         current = current->next;
       }//end while
}//end printList

template<class Type>
void DoublyLinkedList<Type>::printBack() const
{
     DLNode<Type> *current; //pointer to traverse the list
    current = last;  //set current to point to the last node
    while (current != NULL)
     {
           cout << current->info << "  ";
           current = current->back;
     }//end while
}//end reversePrint
```

A search can be conducted on a doubly linked list object using the search() member function. This function takes the item to be searched as an argument and returns true if it finds the item in the list, otherwise it returns false.

```
template<class Type>
bool DoublyLinkedList<Type>::search(const Type&
searchItem)  const
{
   bool found = false;
   DLNode<Type> *current; //pointer to traverse the list
   current = first;
   while (current != NULL && !found)
      if (current->info >= searchItem)
         found = true;
      else
         current = current->next;
   if (found)
     found = (current->info == searchItem);
       //test for equality
   return found;
}//end search
```

Inserting nodes in a double linked list.

Adjusting the node's pointer variables next and previous can do the insert operation. It is easy to add a node at the front of the list and at the rear of the list. The following picture shows the steps involved in inserting a node into a doubly linked list.

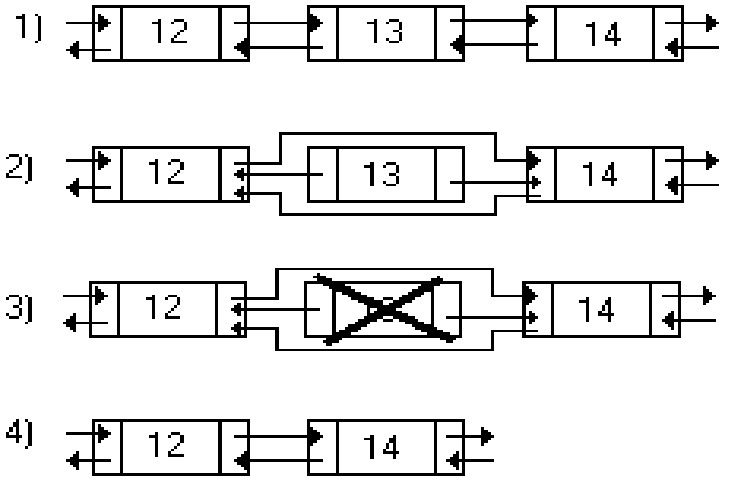



```
template<class Type>
void DoublyLinkedList<Type>::insertNode(const Type&
insertItem)
{
    DLNode<Type> *current;  //pointer to traverse the list
    DLNode<Type> *trailCurrent;
      //pointer just before current
    DLNode<Type> *newNode; //pointer to create a node
    bool found;
    newNode = new DLNode<Type>; //create the node
    assert(newNode != NULL);
    newNode->info = insertItem;
      //store the new item in the node
    newNode->next = NULL;
    newNode->back = NULL;
    if(first == NULL)
   //if the list is empty, newNode is the only node
    {
      first = newNode;
      last = newNode;
      count++;
    }
```

```
    else
    {
        found = false;
        current = first;

        while (current != NULL && !found) //search the list
            if (current->info >= insertItem)
                found = true;
            else {
                trailCurrent = current;
                current = current->next;
            }

        if (current == first) //insert newNode before first
        {
            first->back = newNode;
            newNode->next = first;
            first = newNode;
            count++;
        }

    else {
        //insert newNode between trailCurrent and current
            if (current != NULL) {
                trailCurrent->next = newNode;
                newNode->back = trailCurrent;
                newNode->next = current;
                current->back = newNode;
            }
            else {
                trailCurrent->next = newNode;
                newNode->back = trailCurrent;
                last = newNode;
            }
            count++;
        }//end else
    }//end else
}//end insertNode
```

Deleting nodes from a double linked list.

The delete operation on a double linked list separates a node from the list and returns the memory occupied by the node to the free memory area list of the operating system. The desired node is separated from the list by adjusting the next and previous variables in its adjacent nodes. The next and previous variables of the target node are made NULL. The following picture shows the steps involved in deleting a node from a doubly linked list.




```
template<class Type>
void DoublyLinkedList<Type>::deleteNode(const Type&
deleteItem)
{
  DLNode<Type> *current; //pointer to traverse the list
  DLNode<Type> *trailCurrent;
    //pointer just before current
  bool found;
  if (first == NULL)
    cout << "Cannot delete from an empty list." << endl;
  else
    if (first->info == deleteItem)
```

```
    //node to be deleted is the first node
     {
        current = first;
        first = first->next;
        if (first != NULL)
            first->back = NULL;
        else
            last = NULL;
        count--;
        delete current;
     }
     else
     {
        found = false;
         current = first;
        while (current != NULL && !found)  //search the list
           if (current->info >= deleteItem)
               found = true;
           else
               current = current->next;
        if (current == NULL)
        cout << "The item to be deleted is not in the list."
               << endl;
else
    if (current->info == deleteItem)
           //check for equality
         {
           trailCurrent = current->back;
           trailCurrent->next = current->next;
           if (current->next != NULL)
              current->next->back = trailCurrent;
           if (current == last)
              last = trailCurrent;

           count--;
           delete current;
         }
         else
           cout << "The item to be deleted is not in list."
```

```
                << endl;
        }//end else
}//end deleteNode

template<class Type>
DoublyLinkedList<Type>::~DoublyLinkedList()
{
cout << "Definition of the destructor is left as “
       <<”an exercise." << endl;
cout << "See Programming Execrise 9." << endl;
}
```

12.6. Summary:

Linked lists have several advantages over arrays. Elements can be inserted into linked lists indefinitely, while an array will eventually either fill up or need to be resized. It is an expensive operation toresize an array. Dynamic insertions and deletions are not easy to do with arrays.

On the other hand, arrays allow random access, while linked lists allow only sequential access to elements. Singly-linked lists, in fact, can only be traversed in one direction. This makes linked lists unsuitable for applications where it's useful to look up an element by its index quickly, such as heapsort. Sequential access on arrays is also faster than on linked lists on many machines due to locality of reference and data caches. Linked lists receive almost no benefit from the cache.

Another disadvantage of linked lists is the extra storage needed for references, which often makes them impractical for lists of small data items such as characters or boolean values. It can also be slow.

Double-linked lists require more space per node (unless one uses xor-linking), and their elementary operations are more expensive; but they are often easier to manipulate because they allow sequential access to the list in both directions. In particular, one can insert or delete a node in a constant number of operations given only that node's address. Some algorithms require access in both directions.

Circular linked lists are most useful for describing naturally circular structures, and have the advantage of regular structure and being able to traverse the list starting at any point. They also allow quick access to the first and last records through a single pointer (the address of the last element). Their main disadvantage is the complexity of iteration.

12.7. Technical Terms:

Data structure
A data structure is a way of storing data in a computer so that it can be used efficiently.

Linked list
A linked list is one of the fundamental data structures used in computer programming. It consists of a sequence of nodes, each containing arbitrary data fields and one or two references ("links") pointing to the next and/or previous nodes.

Double Linked List
It is a linear linked list with two links in each node. One link called ‘next’ points to the next node in the list, while the other link called ‘previous’ points to the previous node in the list.

Pointer: The memory address of a variable or object.

Pointer Variable: A variable that contains a memory address.

Head or first
Location, where the address of the first node in the list is stored.

Link
A component used to store the address of the next node in a linked list.

Nodes: Components of a linked list.

12.8. Model Questions:

1. What is a linked list? Explain various operations on list?
2. Write the ADT for linear linked list?
3. Explain the insert and delete operations on a single linked list?
4. How do you insert a node into a doubly linked list?
5. How do you delete a specific node from a doubly linked list?

12.9. References:

"**Data Structures, Algorithms, and Applications in C++**" by SAHNI

"Let us C++" by Yeshavanth kanethkar

"ANSI C++" by Blaguruswamy

AUTHOR:

Y. VENKATESWARA RAO, M.C.A.,
Lecturer,
Dept.Of Computer Science,
JKC College, GUNTUR

Lesson 13: Trees and Graphs

Objectives

In this chapter you will:

- Learn about Trees
- Learn about varieties of Trees
- Learn how to implement a Tree data structure
- Examine different Tree traversal methods
- Discover Tree applications
- Learn about Graphs
- Examine representation of Graphs in memory
- Learn about Graph traversal methods

Structure of the Lesson

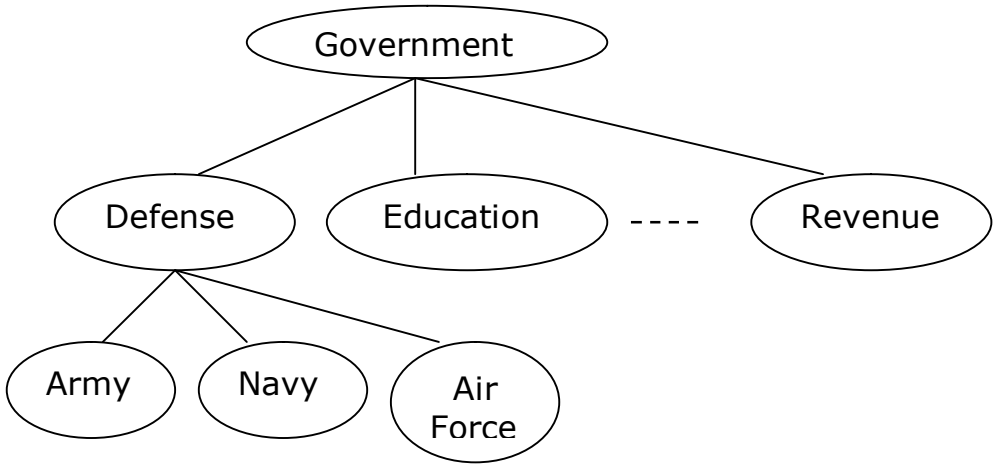
13.1.Introduction

A **tree** is a non-linear data structure in which elements are represented as nodes and are linked together in hierarchical fashion. A tree has the ability to grow and expand, and is therefore a dynamic, flexible, and open-ended system.

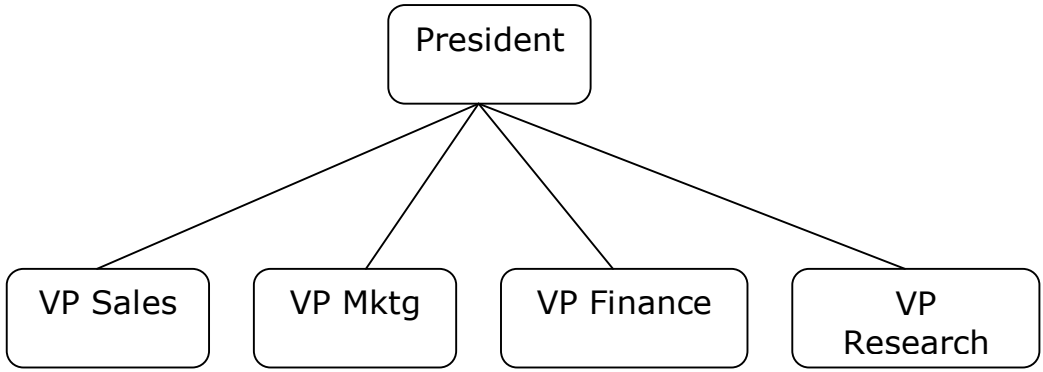
Definition: A tree t is a finite nonempty set of elements. One of these elements is called the **root**, and the remaining elements (if any) are partitioned into trees, which are called the **subtrees** of t.

A tree is drawn with each element represented as a node. The root node is drawn at the top, and its subtrees are drawn below. An edge (line) is drawn from the tree root to the roots of its subtrees (if any). The roots of the subtrees are called **children** of the root node, and root node is termed as their **parent**. Children of the same parent are called **siblings**. Each subtree is drawn similarly with its root at the top and its subtrees below. In a tree, a node with no child nodes is called **leaf**. The number of children of a node is called the **degree** of that node. Every node in a tree has a **level**. By definition the tree root is at level 1; Its children (if any) are at level 2; Their children (if any) are at level 3; and so on.

Trees are very much useful to represent hierarchical data. Hierarchical data has ancestor-descendant, superior-subordinate, or whole-part relationship among data elements. For example the subdivisions of a government can be shown as in the following tree.



The hierarchical administrative structure of a corporation can be shown as following tree.



13.2. The Tree Data structure.

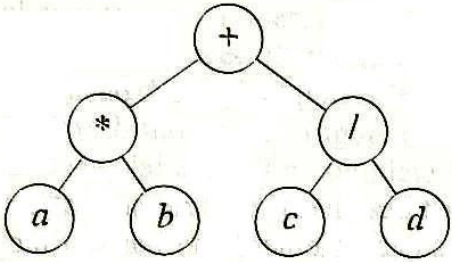
Binary Tree

A Binary Tree t is a finite (may be empty) collection of elements, with one element designated as **root**, and the remaining elements partitioned into two binary trees, which are called left and right subtrees of t.

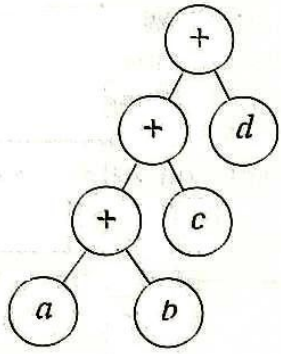
The subtle differences between a tree and a binary tree are given in the following table.

Tree	Binary Tree
A tree cannot be empty	A binary tree can be empty
Each element can have any number of subtrees.	Each element can have exactly two subtrees.
The subtrees are unordered.	The subtrees are ordered.

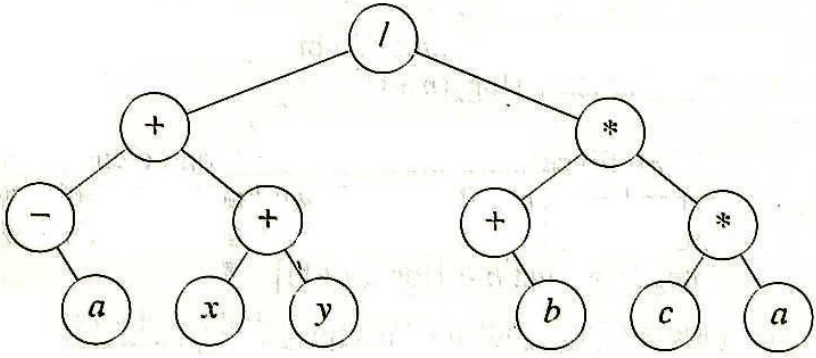
Binary trees are drawn similar to trees, with root node at the top. An example for a binary tree is expression tree. Expression trees are used in generation of optimal computer code to evaluate an expression. The following figures show sample expression trees.



(a) $(a * b) + (c / d)$



(b) $((a + b) + c) + d)$



(c) $((-a) + (x + y)) / ((+b) * (c * a))$

Binary Tree Properties

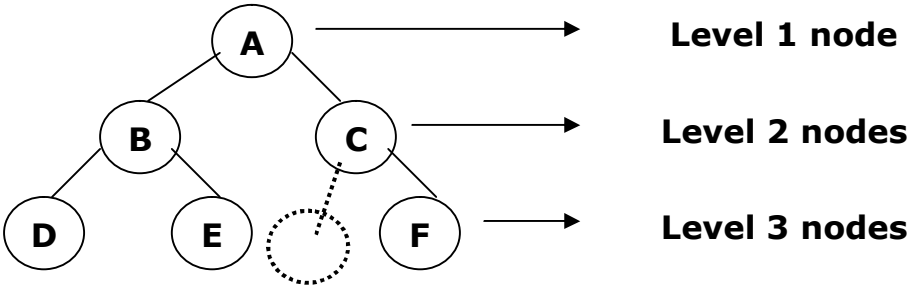
- The drawing of every binary tree with n (n>0) elements has exactly (n-1) edges.
- The number of levels in a binary tree is called its **depth** or **height**.
- A binary tree of height h, $h \geq 0$, has atleast h and at most 2^h-1 elements in it.
- The height of a binary tree that contains n, $n \geq 0$, elements is at most n and at least $\log_2(n+1)$.
- A binary tree with height h, and contains exactly 2^h-1 elements is called a **Full Binary Tree**.
- The nodes in full binary tree are numbered sequentially, starting at 1, from level 1 to level h, and from left to right in each level.
- A binary tree with maximum posible number of nodes at each level, except possibly the last is called a **complete binary tree**.
- In a binary tree the maximum possible number of nodes at level k is $2^{(k-1)}$.
- For an element numbered k, in a complete binary tree, if k=1, then it is the root element. if k>1, then its parent has been assigned the number (int) (k/2). Its left child is numbered 2k (no left child if 2k > n), and right child is numbered 2k+1 (no right child if 2k+1 > n , where n is maximum number of nodes).

Binary Search Tree (BST)

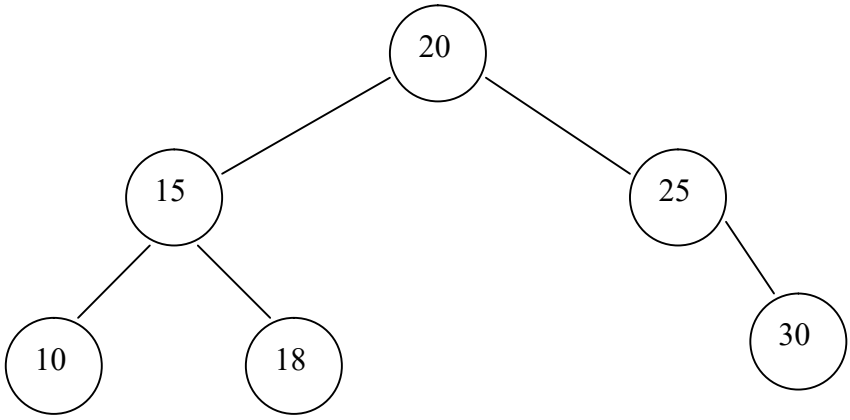
A BST is a binary tree with the following properties:

1. Every element has a key (or value). All keys are distinct.
2. The keys (if any) in the left subtree of the root are smaller than the key in the root.
3. The keys (if any) in the right subtree of the root are larger than the key in the root.
4. The left and right subtrees of the root are also binary search trees.

An example of a binary tree is shown below. (A circle with dotted line indicates missing node.)

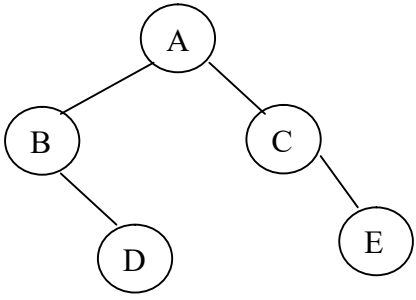


An example of a binary search tree in which elements have distinct keys is shown below.



13.3. Formula-based representation

The formula based representation for binary trees makes use of the last property stated above for binary trees. In this method, the binary tree is represented in an array by storing each element at the array position corresponding to the number assigned to it. This representation is more suitable to either full or complete binary trees. For other binary trees it is inefficient if a number of nodes are missing from the tree. In this method a binary tree with n elements may require an array of size up to 2^n-1 for its representation. The following picture shows a binary tree and its formula-based representation.

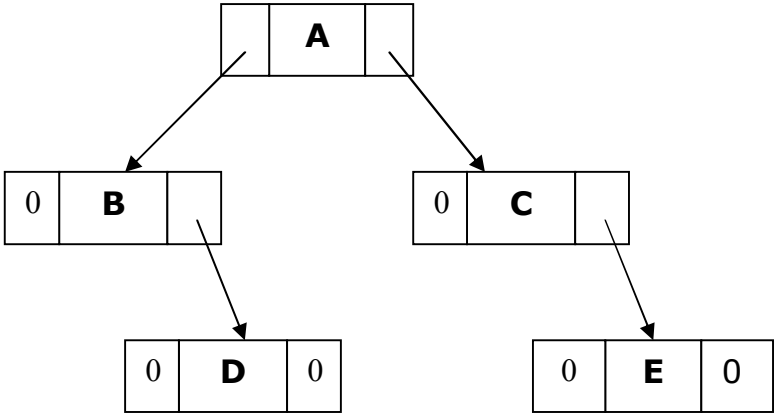


A	B	C		D		E
1	2	3	4	5	6	7

13.4. Linked representation

The Linked representation is a popular way to store binary trees in memory. This representation uses links or pointers. A node that has exactly two link fields represents each element. The links are called *left_child* and *right_child*. In addition to these two link fields, each node has a field has a field named *data*. An edge in the drawing of a binary tree is represented by, a pointer from the parent node to the child node. This pointer is placed in the appropriate link field of the parent node. Since an n-element binary tree has exactly n-1 edges, (n+1) link fields are set to zero or NULL. The following is the node class for linked representation of binary trees:

```
template<class T>
class BinaryTreeNode{
   public:
      BinaryTreeNode() { left_child = right_child = 0; }
      BinaryTreeNode(const T&e)   {
            data = e;              left_child = right_child = 0;
       }
    private:
        T data;
         BinaryTreeNode<T> *left_child, // left subtree
                                *right_child; // right subtree.
};
```



Linked Representation of a binary tree

Common binary tree operations: Some common operations on binary trees are:

- Determine its height.
- Determine the number of elements in it.
- Make a copy.
- Delete the tree.
- Traverse and list the nodes in a tree.
- Search for a specific node in a tree.

The above said operations can be performed, by traversing the binary tree in a systematic manner. In a binary tree **traversal**, each element is **visited** exactly once. During this visit the necessary action regarding this node is taken. There are four common ways to traverse a binary tree. They are:

- Preorder
- Inorder
- Postorder
- Level order

The first three traversal methods are described in the following recursive algorithms and procedures.

Algorithm for **preorder** traversal:

step1: Visit the root node.

step2: Traverse the left subtree in preorder.

setp3: Traverse the right subtree in preorder.

Recursive implementation of the above algorithm:

```
template <class T>
void preorder(BinaryTreeNode<T> *t)
{ // preorder traversal of *t.
   if( t) {
     visit(t);
     preorder(t->left_child);   // start preorder traversal of left subtree.
     preorder(t->right_child);//start preorder traversal of right subtree.
   }
}
```

Algorithm for **Inorder** traversal:

step1: Traverse the left subtree in inorder.

step2: Visit the root node.

setp3: Traverse the right subtree in inorder.

Recursive implementation of the above algorithm:

```
template <class T>
void inorder(BinaryTreeNode<T> *t)
{ // inorder traversal of *t.
   if( t) {
     inorder(t->left_child);   // start inorder traversal of left subtree.
     visit(t);
     inorder(t->right_child);//start inorder traversal of right subtree.
   }
}
```

Algorithm for **postorder** traversal:

step1: Traverse the left subtree in postorder.

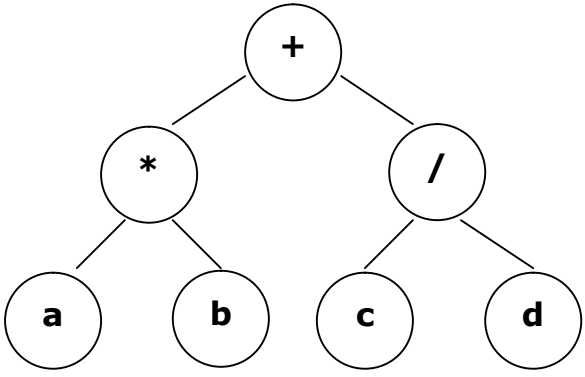
setp2: Traverse the right subtree in postorder.

step3: Visit the root node.

Recursive implementation of the above algorithm:

```
template <class T>
void postorder(BinaryTreeNode<T> *t)
{ // postorder traversal of *t.
   if( t) {
     postorder(t->left_child);   // start inorder traversal of left subtree.
     postorder(t->right_child);//start inorder traversal of right subtree.
     visit(t);
   }
}
```

The visit() function in the above implementations defines the necessary action to be taken on the nodes. Its simplest implementation is to display the data at the node. In the preorder, inorder, and postorder traversal methods the left subtree is traversed before the right subtree. The difference in these traversals is in the time at which a node is visited. In preorder each node is visited before its left and right subtree nodes are visited. In inorder traversal, each node is visited after the left subtree nodes and are visited and before the right subtree nodes. In postorder traversal, each node is visited after both the left and right subtree nodes are visited in that order. For the expression tree shown below the preorder, inorder and postorder traversal methods give the prefix, infix and postfix notations of the expression represented by the tree.



Preorder: +*ab/cd

Inorder: a*b+c/d

Postorder: ab*cd/+

The infix form of an expression is the form in which we normally write an expression. In this form each binary operator appears between its operands. In the prefix form each operator comes immediately before the prefix from of its operands. The operands appear in left to right order. In postfix notation each operator comes immediately after the postfix form of its operands. The operands appear in left to right order.

Level order traversal: In a level order traversal of a binary tree, the elements are visited by level from top to bottom. Within each level, elements are visited from left to right. The following function shows an implementation of the level order traversal of a binary tree.

```
// level order traversal.
template <class T>
void levelorder(BinaryTreeNode<T> *t)
{ // levelorder traversal of *t.
   LinkedQueue<BinaryTreeNode<T>*> q;
   while( t) {
     visit(t);
     if (t->left_child) q.add(t->left_child);
     if (t->right_child) q.add(t->right_child);
     // get next node to visit.
      try { q.delete(t); } catch(OutOfBounds) {return;}
   }
}
```

The space complexity of each of the four traversal programs is O(n) and time complexity is $\Theta(n)$, where n is the number of nodes in the binary tree.

13.5. Binary Tree ADT & Implementation

Having some understanding of binary tree, we can specify an ADT for binary tree as below.

AbstractDataType BinaryTree{

instances: collection elements; if not empty, the collection is partitioned into a root, left subtree, and right subtree; each subtree is also binary tree.

operations:

Create(): Create an empty binary tree.

IsEmpty(): Return *true* if empty, *false* otherwise.

MakeTree(root, left, right): create a binary tree with root as the root element, left and right subtrees.

PreOrder(): Do preorder traversal of the binary tree.

Inorder(): Do inorder traversal of the binary tree.

Postorder(): Do postorder traversal of the binary tree.

LevelOrder(): Do level order traversal of the binary tree.

}

The following program gives the implementation of the binary tree ADT.

```
#ifndef BinaryTree_
#define BinaryTree_
int _count;

#include<iostream.h>
#include "lqueue.h"
#include "btnode2.h"
#include "xcept.h"

template<class E, class K> class BSTree;
template<class E, class K> class DBSTree;

template<class T>
class BinaryTree {
  friend BSTree<T,int>;
  friend DBSTree<T,int>;
  public:
    BinaryTree() {root = 0;};
    ~BinaryTree(){};
    bool IsEmpty() const
      {return ((root) ? false : true);}
    bool Root(T& x) const;
    void MakeTree(const T& element,
        BinaryTree<T>& left, BinaryTree<T>& right);
    void BreakTree(T& element, BinaryTree<T>& left,
                BinaryTree<T>& right);
    void PreOrder(void(*Visit)(BinaryTreeNode<T> *u))
        {PreOrder(Visit, root);}
```

```
    void InOrder(void(*Visit)(BinaryTreeNode<T> *u))
        {InOrder(Visit, root);}
    void PostOrder(void(*Visit)(BinaryTreeNode<T> *u))
        {PostOrder(Visit, root);}
    void LevelOrder(void(*Visit)(BinaryTreeNode<T> *u));
    void PreOutput() {PreOrder(Output, root); cout << endl;}
    void InOutput() {InOrder(Output, root);
     cout << endl;}
    void PostOutput() {PostOrder(Output, root);
      cout << endl;}
    void LevelOutput() {LevelOrder(Output);
      cout << endl;}
    void Delete() {PostOrder(Free, root); root = 0;}
    int Height() const {return Height(root);}
    int Size()
      {_count = 0; PreOrder(Add1, root); return _count;}
  private:
    BinaryTreeNode<T> *root;  // pointer to root
    void PreOrder(void(*Visit)
      (BinaryTreeNode<T> *u), BinaryTreeNode<T> *t);
    void InOrder(void(*Visit)
      (BinaryTreeNode<T> *u), BinaryTreeNode<T> *t);
    void PostOrder(void(*Visit)
      (BinaryTreeNode<T> *u), BinaryTreeNode<T> *t);
    static void Free(BinaryTreeNode<T> *t) {delete t;}
    static void Output(BinaryTreeNode<T> *t)
            {cout << t->data << ' ';}
    static void Add1(BinaryTreeNode<T> *t) {_count++;}
    int Height(BinaryTreeNode<T> *t) const;
};
```

```
template<class T>
bool BinaryTree<T>::Root(T& x) const
{// Return root data in x.
 // Return false if no root.
   if (root) {x = root->data;
              return true;}
   else return false;  // no root
}

template<class T>
void BinaryTree<T>::MakeTree(const T& element,
          BinaryTree<T>& left, BinaryTree<T>& right)
{// Combine left, right, and element to make new tree.
 // left, right, and this must be different trees.
   // create combined tree
   root = new BinaryTreeNode<T>
              (element, left.root, right.root);
   // deny access from trees left and right
   left.root = right.root = 0;
}

template<class T>
void BinaryTree<T>::BreakTree(T& element,
       BinaryTree<T>& left, BinaryTree<T>& right)
{// left, right, and this must be different trees.
   // check if empty
   if (!root) throw BadInput(); // tree empty
```

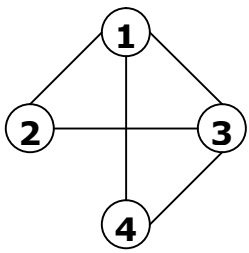
```
   // break the tree
   element = root->data;
   left.root = root->LeftChild;
   right.root = root->RightChild;
   delete root;
   root = 0;
}
template<class T>
void BinaryTree<T>::PreOrder(
         void(*Visit)(BinaryTreeNode<T> *u),  BinaryTreeNode<T> *t)
{// Preorder traversal.
   if (t) {
            Visit(t);
            PreOrder(Visit, t->LeftChild);
            PreOrder(Visit, t->RightChild);
         }
}
template <class T>
void BinaryTree<T>::InOrder(
      void(*Visit)(BinaryTreeNode<T> *u),  BinaryTreeNode<T> *t)
{// Inorder traversal.
   if (t) {
           InOrder(Visit, t->LeftChild);
            Visit(t);
            InOrder(Visit, t->RightChild);
         }
}
```

```
template <class T>
void BinaryTree<T>::PostOrder( void(*Visit)(BinaryTreeNode<T> *u),
     BinaryTreeNode<T> *t)  {// Postorder traversal.
   if (t) {
         PostOrder(Visit, t->LeftChild);
         PostOrder(Visit, t->RightChild);
         Visit(t);
         }
}
template <class T>
void BinaryTree<T>::LevelOrder( void(*Visit)(BinaryTreeNode<T> *u))
  {// Level-order traversal.
   LinkedQueue<BinaryTreeNode<T>*> Q;
   BinaryTreeNode<T> *t;
   t = root;
   while (t) {
      Visit(t);
      if (t->LeftChild) Q.Add(t->LeftChild);
      if (t->RightChild) Q.Add(t->RightChild);
      try {Q.Delete(t);
      }
      catch (OutOfBounds) {return;}
      }
}

template <class T>
int BinaryTree<T>::Height(BinaryTreeNode<T> *t) const
{// Return height of tree *t.
   if (!t) return 0;              // empty tree
   int hl = Height(t->LeftChild);  // height of left
```

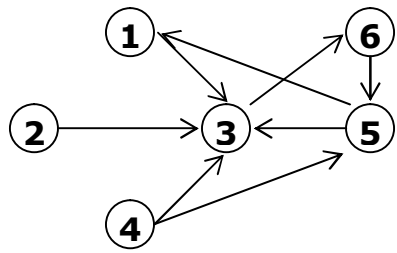
```
   int hr = Height(t->RightChild); // height of right
   if (hl > hr) return ++hl;
   else return ++hr;
}
#endif
```

13.6. Graph Data Structure

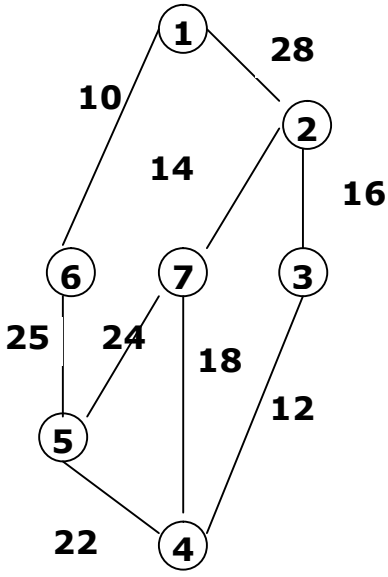
A graph is a collection of nodes, pairs of which are joined by lines or edges. A more formal definition can be given as:
Definition: A **graph** G = (V,E) is an ordered pair of finite sets V and E. The elements of V are called **vertices** or **nodes** or points. The elements of E are called **edges**. Each edge in E joins two different vertices of V and is denoted by the ordered pair (i,j), where I and j are the two vertices.
A graph is displayed with nodes as circles and edges as lines. The edges may have a orientation. An edge with an orientation is called a **directed edge**. An **undirected edge** has no orientation. If all the edges in a graph are directed then it is called a directed graph or **digraph**. Two vertices i and j are called adjacent if and only if there is an edge from vertex i to vertex j. The edge (i,j) is incident on vertices i and j. When weights have been assigned to edges, then that graph is called a weighted graph. Some examples of graphs are shown below:



graph



digraph



weighted graph

Path: a sequence of vertices P = i_1, i_2,… ,i_k is an i_1 to i_k path in the graph or digraph G = (V,E) if and only if the edge (i_j,i_{j+1}) is in E for every j, $1 \leq j < k$.

Simple path: It is a path in which all vertices, except possibly the first and last, are different.

Length of a path: The length of a path is the number of edges involved in that path.

Cycle: A cycle is a simple path with the same start and end vertex.

Subgraph: A graph H is a subgraph of another graph G if and only if its vertex and edge sets are subsets of those of G.

Connected graph: A graph G is connected if and only if there is a path between every pair of vertices in G.

Note: A connected undirected graph that contains no cycles is a tree.

Spanning tree: A subgraph of G that contains all the vertices of G and is a tree is a spanning tree of G.

Some properties of graphs:

1) A connected graph with n vertices must have at least n-1 edges.
2) Let G be an undirected graph. The degree d_i of vertex i is the number of edges incident on vertex i.
3) An n-vertex graph with n(n-1)/2 edges is a complete graph.
4) Let G be a digraph. The in-degree d_i^{in} of vertex i is the number of edges incident to i. The out-degree d_i^{out} of vertex i is the number of edges incident from this vertex.
5) A complete digraph with n vertices contains exactly n(n-1) directed edges.

The ADTs *Graph* and *Digraph*

The abstract data type Graph refers to undirected graphs. The abstract data type Digraph refers to digraphs. The below listing gives the ADTs Graph and Digraph.

AbstractDataType Graph{
  instances
    a set V of vertices and a set E of edges
  operations
    Create(n): create an undirected graph with n vertices and no edges

    Exist(i,j): return *true* if edge (i,j) exists, *false* otherwise

    Edges(): return the number of edges in the graph
    Vertices(): return the number of vertices in the graph

    Add(i,j): add the edge (i,j)

    Delete(i,j): delete the edge (i,j)

    Degree(i): return the degree of vertex i.
}

AbstractDataType DiGraph{
  instances
    a set V of vertices and a set E of edges
  operations
    Create(n): create a directed graph with n vertices and no edges

    Exist(i,j): return *true* if edge (i,j) exists, *false* otherwise

    Edges(): return the number of edges in the graph

    Vertices(): return the number of vertices in the graph

    Add(i,j): add the edge (i,j) to the graph

    Delete(i,j): delete the edge (i,j)

    InDegree(i): return the in-degree of vertex i.

    OutDegree(i): return the out-degree of vertex i
}

13.7. Adjacency Matrix and Adjacency Lists.

The most frequently used representation schemes for graphs and digraphs are adjacency based: adjacency matrices, and adjacency lists.

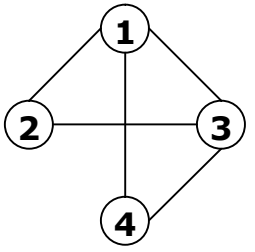
Adjacency Matrix
The adjacency matrix of an n-vertex graph G = (V,E) is an n x n matrix A. Each element of A is either zero or one. We shall assume that V = {1,2,…,n}. If G is an undirected graph, then the elements of A are defined as follows:

$$A(i,j) = \begin{cases} 1 & \text{if } (i,j) \in E \text{ or } (j,i) \in E \\ 0 & \text{otherwise} \end{cases}$$

If G is a digraph, then the elements of A are defined as follows:

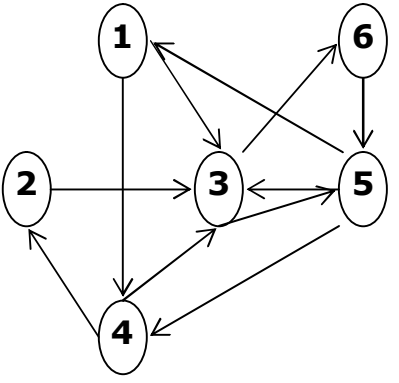
$$A(i,j) = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

The adjacency matrices for two graphs are as shown here.



Adjacency matrix:

	1	2	3	4
1	0	1	1	1
2	1	0	1	0
3	1	1	0	1
4	1	0	1	0



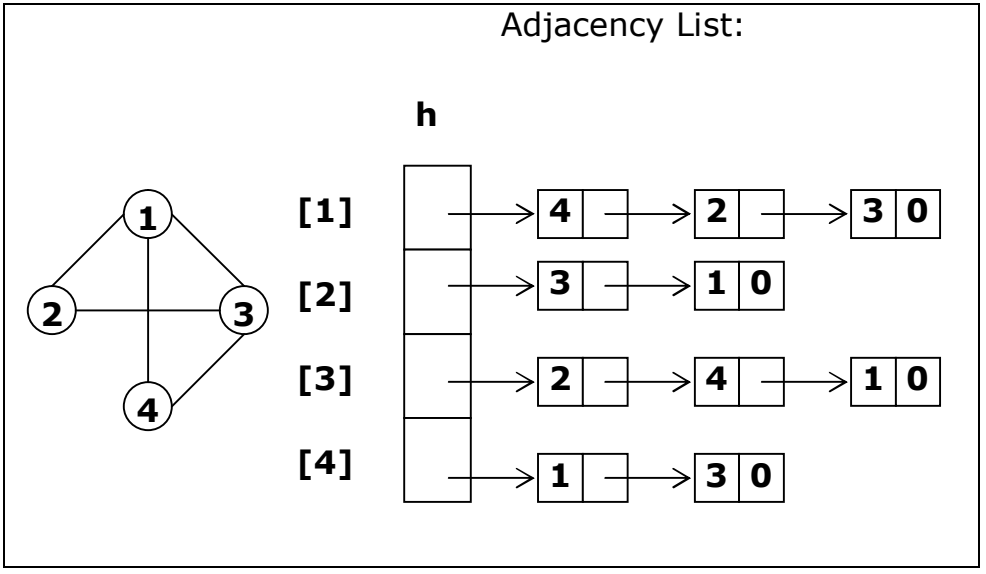
Adjacency matrix:

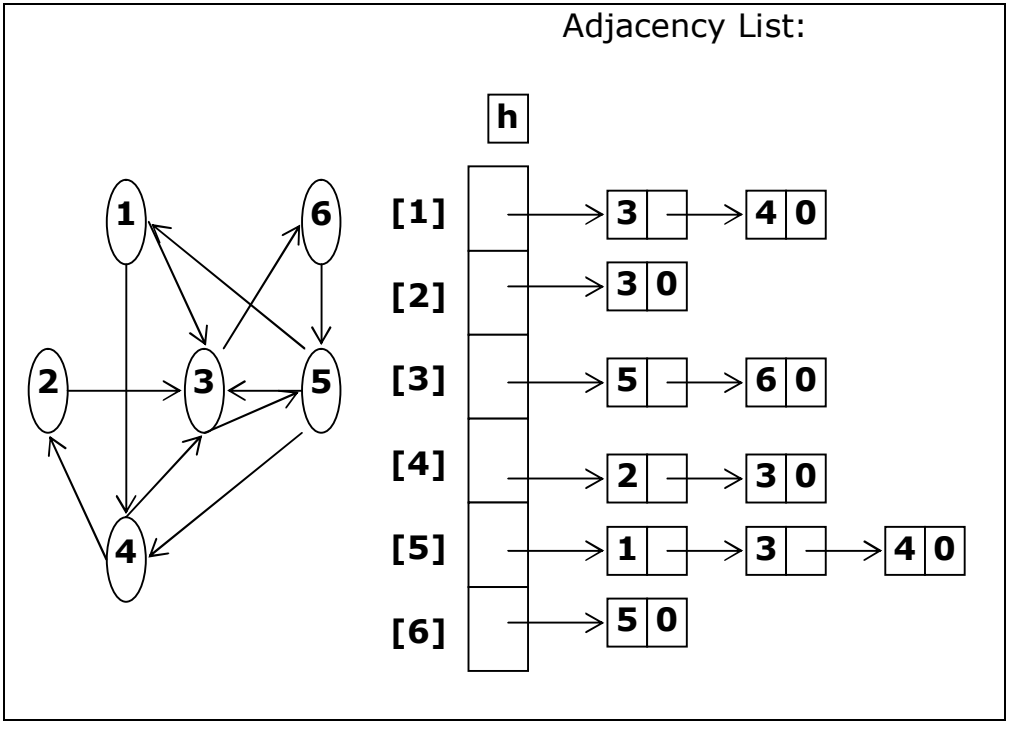
	1	2	3	4	5	6
1	0	0	1	1	0	0
2	0	0	1	0	0	0
3	0	0	0	0	1	1
4	0	1	1	0	0	0
5	1	0	1	1	0	0
6	0	0	0	0	1	0

The n x n adjacency matrix A may be mapped into a array of the same size or of size (n+1) x (n+1) of type int using the mapping A(i,j) = A[i][j], where $1 \leq i \leq n$, and $1 \leq j \leq n$.

Adjaceny Lists

In the case of adjaceny lists (or linked-adjacency lists), each adjacency list is maintained as a chain. The adjacency lists of the above given two grapsh are as shown.


Adjacency List:
h
1
2
3
4
[1]
[2]
[3]
[4]
4
2
3 0
3
1 0
2
4
1 0
1
3 0


Adjacency List:
h
1
6
2
3
5
4
[1]
[2]
[3]
[4]
[5]
[6]
3
4 0
3 0
5
6 0
2
3 0
1
3
4 0
5 0

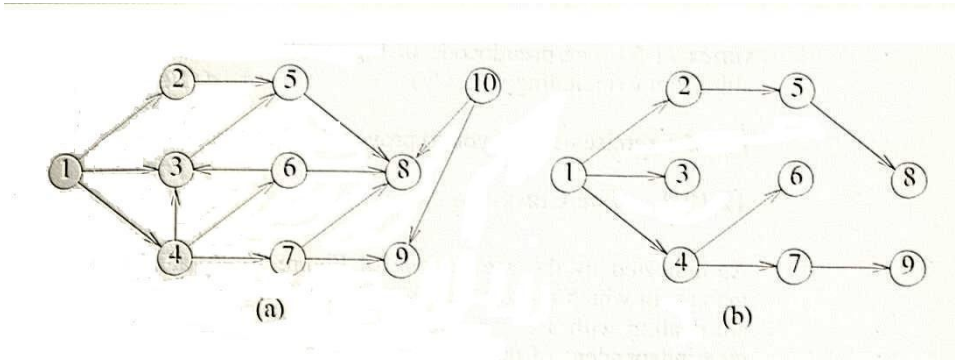
13.8. Breadth-First and Depth-First Search

Many operations on graph require traversing its nodes. There are two standard ways to do this. These are known as search methods. They are Breadth-First search and Depth-First Search. Although both methods are popular, Depth-First search is used frequently.

Breadth-First Search (BFS)

This method proceeds by starting at a vertex and identifying all vertices reachable from it. i.e. identifying all adjacent vertices to it and repeating this procedure from each such vertex in that order until all the vertices are visited. The queue data structure is used to perform this search. The tree resulting from this search is called Breadth first search spanning tree. The following is the pseudo code for **BFS**.

```
//Breadth first search beginning at vertex v.
Label vertex v as reached;
Initialize Q to be a queue with only v in it;
while(Q is not empty)
{
    Delete a vertex w from the queue;
    Let u be a vertex (if any) adjacent from w;
      while(u){
        if (u has not been labeled)
        {
            Add u to the queue;
            Label u as reached;
            u = next vertex that is adjacent from w;
        }
      }
}
```



(a) Directed Graph (b)Breadth-First Search Tree of Graph in (a).

Depth-First Search (DFS)

This is an alternative to BFS. Starting at a vertex v, the DFS proceeds as follows: First the vertex v s marked as *visited*, and then an *unreached* vertex u adjacent from v is selected. If such a vertex does not exist, the search terminates. If u exists, the DFS is now initiated from u. When this search is completed, another vertex adjacent from v is selected, and the process continues until no such un visited vertex exists. The tree obtained from DFS is called DFS spanning tree. The pseudo code for DFS is given below.

```
//Depth first search beginning at vertex v.
Label vertex v as reached;
Initialize S to be a stack with only v on its top;
while(S is not empty)
{
    pop a vertex w from the stack S;
    Let u be a vertex (if any) adjacent from w;
        if (u has not been labeled)
        {
            push u on to the stack S;
            Label u as reached;
         }
     }
}
```

13.9. Summary

In computer science, a **tree** is a widely-used computer data structure that emulates a tree structure with a set of linked nodes. It is a special case of a graph. A tree is considered as a recursive structure that usually maps an ordered set of data from an internal definition to some *data space*. Each node in a tree has zero or more **child nodes**, which are below it in the tree. A node that has a child is called the child's **parent node**. A child has at most one parent; The topmost node in a tree is called the **root node**. Being the topmost node, the root node will not have parents. Nodes at the bottom most level of the tree are called **leaf nodes**. Since they are at the bottom most level, they will not have any children. A **binary tree** is a **rooted** tree in which every node has at most two children. A **full binary tree** is a tree in which every node has zero or two children. Also known as a **proper binary tree**. A **perfect binary tree** is a full binary tree in which all **leaves** (vertices with zero children) are at the same **depth** (distance from the **root**, also called **height**).

Pre-order, in-order, and post-order traversal visit each node in a tree by recursively visiting each node in the left and right subtrees of the root. If the root node is visited before its subtrees, this is preorder; if after, postorder; if between, in-order. In-order traversal is useful in binary search trees, where this traversal visits the nodes in increasing order.
a **graph** is an abstract data type (ADT) that consists of a set of nodes and a set of edges that establish relationships (connections) between the nodes.

A graph G is defined as follows: G=(V,E), where V is a finite, non-empty set of vertices and E is a set of edges (links between pairs of vertices). When the edges in a graph have no direction, the graph is called undirected, otherwise called directed. In practice, some information is associated with each node and edge.

An adjacency list associates each node with an array of incident edges. If no information is required to be stored in edges, only in nodes, these arrays can simply be pointers to other nodes and thus represent edges with little memory requirement. An advantage of this approach is that new nodes can be added to the graph easily, and they can be connected with existing nodes simply by adding elements to the appropriate arrays. A disadvantage is that determining whether an edge exists between two nodes requires O(n) time, where n is the average number of incident edges per node.

An alternative way is to keep a square matrix (a two-dimensional array) *M* of boolean values (or integer values, if the edges also have weights or costs associated with them). The entry $M_{i,j}$ then specifies whether an edge exists that goes from node *i* to node *j*. An advantage of this approach is that finding out whether an edge exists between two nodes becomes a trivial constant-time memory look-up. Similarly, adding or removing an edge is a constant-time memory access.

In depth-first order, we always attempt to visit the node farthest from the root that we can, but with the caveat that it must be a child of a node we have already visited. Unlike a depth-first search on graphs, there is no need to remember all the nodes we have visited, because a tree cannot contain cycles. Preorder, in-order, and postorder traversal are all special cases of this. Contrasting with depth-first order is breadth-first order, which always attempts to visit the node closest to the root that it has not already visited.

13.10. Technical Terms

Tree

A **tree** is a non-linear data structure in which elements are represented as nodes and are linked together in hierarchical fashion.

Root

A **root node** is a specially chosen node in a tree data structure at which all operations on the tree begin

Degree

The number of children of a node is called the **degree** of that node.

Leaf

A node with no child nodes is called **leaf**.

Binary Tree

A **binary tree** is a **rooted** tree in which every node has at most two children.

Graph

A graph is a collection of nodes, pairs of which are joined by lines or edges.

13.11. Model Questions

1. Define a Tree. Describe its properties.
2. Describe Tree Traversal techniques.
3. Implement a Binary Tree and its operations using C++.
4. Discuss different Memory Representations of Trees.
5. Define a Graph. Discuss its Memory Representations.
6. State the difference and similarity between a Tree and a Graph.
7. Discuss DFS and BFS procedures.
8. Implement Graph search algorithms using C++;
9. Write the ADTs for Tree and Graph.

13.12. References