Notes

Subject: Programming in C++

Class: I B.Sc. (CS) (Semester –II)

CORE COURSE II

PROGRAMMING IN C++

Objective:

To impart basic knowledge of Programming Skills in C++ language.

Unit I

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

Unit II

Classes and Objects – Constructors and Destructors –Operator Overloading and Type Conversions

Unit III

Inheritance : Extending Classes – Pointers - Virtual Functions and Polymorphism

Unit IV

Managing Console I/O Operations – Working with Files – Templates – Exception Handling

Unit V

Standard Template Library – Manipulating Strings – Object Oriented Systems Development

Text Book

1. Balagursamy E, Object Oriented Programming with C++, Tata McGraw Hill Publications, Sixth Edition, 2013

Reference Books

1. Ashok Kamthane, Programming in C++, Pearson Education,2013.

*****

Unit-I

Introduction

Object oriented Programming

Object oriented Programming is defined as an approach that provides a way of modularizing programs by creating partitioned memory area for both data and functions that can be used as templates for creating copies of such modules on demand. Writing object-oriented programs involves creating classes, creating objects from those classes, and creating applications, which are stand-alone executable programs that use those objects. After being created, classes can be reused over and over again to develop new programs. Thinking in an object-oriented manner involves envisioning program components as objects that belong to classes and are similar to concrete objects in the real world; then, you can manipulate the objects and have them interrelate with each other to achieve a desired result.

Basic Concepts of Object oriented Programming

1. Class

A class is a user defined data type. A class is a logical abstraction. It is a template that defines the form of an object. A class specifies both code and data. It is not until an object of that class has been created that a physical representation of that class exists in memory. When you define a class, you declare the data that it contains and the code that operates on that data. Data is contained in instance variables defined by the class known as data members, and code is contained in functions known as member functions. The code and data that constitute a class are called members of the class.

2. Object

An object is an identifiable entity with specific characteristics and behavior. An object is said to be an instance of a class. Defining an object is similar to defining a variable of any data type. Space is set aside for it in memory.

3. Encapsulation

Encapsulation is a programming mechanism that binds together code and the data it manipulates, and that keeps both safe from outside interference and misuse. C++'s basic unit of encapsulation is the class. Within a class, code or data or both may be private to that object or public. Private code or data is known to and accessible by only another part of the object. That is, private code or data cannot be accessed by a piece of the program that exists outside the object. When code or data is public, other parts of your program can access it even though it is defined within an object. Typically, the public parts of an object are used to provide a controlled interface to the private elements of the object. This insulation of the data from direct access by the program is called data hiding.

4. Data abstraction

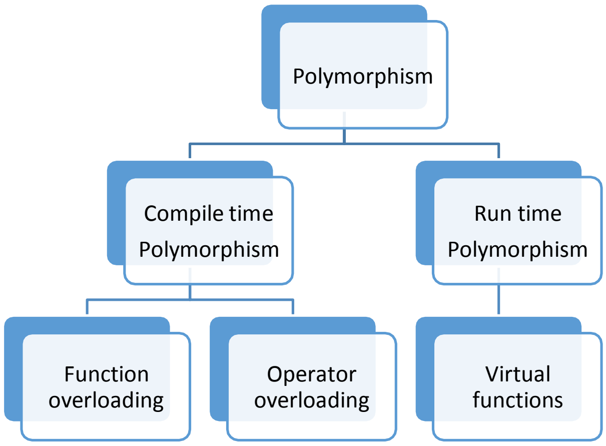
In object oriented programming, each object will have external interfaces through which it can be made use of. There is no need to look into its inner details. The object itself may be made of many smaller objects again with proper interfaces. The user needs to know the external interfaces only to make use of an object. The internal details of the objects are hidden which makes them abstract. The technique of hiding internal details in an object is called data abstraction.

5. Inheritance

Inheritance is the mechanism by which one class can inherit the properties of another. It allows a hierarchy of classes to be build, moving from the most general to the most specific. When one class is inherited by another, the class that is inherited is called the base class. The inheriting class is called the derived class. In general, the process of inheritance begins with the definition of a base class. The base class defines all qualities that will be common to any derived class. . In OOPs, the concept of inheritance provides the idea of reusability. In essence, the base class represent the most general description of a set of traits. The derived class inherits those general traits and adds properties that are specific to that class.

6. Polymorphism

Polymorphism (from the Greek, meaning "many forms") is a feature that allows one interface to be used for a general class of actions. The specific action is determined by the exact nature of the situation. The concept of polymorphism is often expressed by the phrase "one interface, multiple methods." This means that it is possible to design a generic interface to a group of related activities. This helps reduce complexity by allowing the same interface to be used to specify a general class of action. It is the compiler's job to select the specific action as it applies to each situation.



In compile time polymorphism, the compiler is able to select the appropriate function for a particular call at compile time. In C++, it is possible to use one function name for many different purposes. This type of polymorphism is called function overloading. Polymorphism can also be applied to operators. In that case, it is called operator overloading.

In run time polymorphism, the compiler selects the appropriate function for a particular call while the program is running. C++ supports a mechanism known as virtual functions to achieve run time polymorphism.

Need for Object oriented Programming

Object-oriented programming scales very well, from the most trivial of problems to the most complex tasks. It provides a form of abstraction that resonates with techniques people use to solve problems in their everyday life.

Object-oriented programming was developed because limitations were discovered in earlier approaches to programming. There were two related problems. First, functions have unrestricted access to global data. Second, unrelated functions and data, the basis of the procedural paradigm, provide a poor model of the real world.

Benefits of Object oriented Programming

1. Simplicity: Software objects model real world objects, so the complexity is reduced and the program structure is very clear.

2. Modularity: Each object forms a separate entity whose internal workings are decoupled from other parts of the system.

3. Modifiability: It is easy to make minor changes in the data representation or the procedures in an OO program. Changes inside a class do not affect any other part of a program, since the only public interface that the external world has to a class is through the use of methods.

4. Extensibility: adding new features or responding to changing operating environments can be solved by introducing a few new objects and modifying some existing ones.

5. Maintainability: objects can be maintained separately, making locating and fixing problems easier.

6. Re-usability: objects can be reused in different programs.

C++

C++ is an object oriented programming language. It was developed by Bjarne Stroustrup in 1979 at Bell Laboratories in Murray Hill, New Jersey. He initially called the new language "C with Classes." However, in 1983 the name was changed to C++.

C++ is a superset of C. Stroustrup built C++ on the foundation of C, including all of C's features, attributes, and benefits. Most of the features that Stroustrup added to C were designed to support object-oriented programming .These features comprise of classes, inheritance, function overloading and operator overloading. C++ has many other new features as well, including an improved approach to input/output (I/O) and a new way to write comments.

C++ is used for developing applications such as editors, databases, personal file systems, networking utilities, and communication programs. Because C++ shares C's efficiency, much high-performance systems software is constructed using C++.

A Simple C++ Program

```
#include<iostream.h>

#include<conio.h>

int main()

{

 cout<< “Simple C++ program without using class”;

return 0;

}
```

Lines beginning with a hash sign (#) are directives read and interpreted by what is known as the preprocessor. They are special lines interpreted before the compilation of the program itself begins. In this case, the directive #include <iostream.h>, instructs the preprocessor to include a section of standard C++ code, known as header iostream that allows to perform standard input and output operations, such as writing the output of this program to the screen.

The function named main is a special function in all C++ programs; it is the function called when the program is run. The execution of all C++ programs begins with the main function, regardless of where the function is actually located within the code.

The open brace ({) indicates the beginning of main's function definition, and the closing brace (}) indicates its end.

The statement :

cout<< “Simple C++ program without using class”;

causes the string in quotation marks to be displayed on the screen. The identifier cout (pronounced as c out) denotes an object. It points to the standard output device namely the console monitor. The operator << is called insertion operator. It directs the string on its right to the object on its left.

The program ends with this statement:

return 0;

This causes zero to be returned to the calling process (which is usually the operating system). Returning zero indicates that the program terminated normally. Abnormal program termination should be signaled by returning a nonzero value.

The general structure of C++ program with classes is shown as:

1. Documentation Section

2. Preprocessor Directives or Compiler Directives Section

(i) Link Section

(ii) Definition Section

3. Global Declaration Section

4. Class declaration or definition

5. Main C++ program function called main ( )

C++ keywords

When a language is defined, one has to design a set of instructions to be used for communicating with the computer to carry out specific operations. The set of instructions which are used in programming, are called keywords. These are also known as reserved words of the language. They have a specific meaning for the C++ compiler and should be used for giving specific instructions to the computer. These words cannot be used for any other purpose, such as naming a variable. C++ is a case-sensitive language, and it requires that all keywords be in lowercase. C++ keywords are:

asm	auto	bool	break
case	catch	char	class
const	const_cast	continue	default
delete	do	double	dynamic_cast
else	enum	explicit	export
extern	false	float	for
friend	goto	if	inline
int	long	mutable	namespace
new	operator	private	protected
public	register	reinterpret_cast	return
short	signed	sizeof	static
static_cast	struct	switch	template
this	throw	true	try
typedef	typeid	typename	union
unsigned	using	virtual	void
volatile	wchar_t	while	

Identifiers

An identifier is a name assigned to a function, variable, or any other user-defined item. Identifiers can be from one to several characters long.

Rules for naming identifiers:

- Variable names can start with any letter of the alphabet or an underscore. Next comes a letter, a digit, or an underscore.
- Uppercase and lowercase are distinct.
- C++ keywords cannot be used as identifier.

Data types

Data type defines size and type of values that a variable can store along with the set of operations that can be performed on that variable. C++ provides built-in data types that correspond to integers, characters, floating-point values, and Boolean values. There are the seven basic data types in C++ as shown below:

Type	**Meaning**
char(character)	holds 8-bit ASCII characters
wchar_t(Wide character)	holds characters that are part of large character sets
int(Integer)	represent integer numbers having no fractional part
float(floating point)	stores real numbers in the range of about 3.4x10–38to 3.4x10 38,with a precision of seven digits.

double(Double floating point)	Stores real numbers in the range from 1.7x10 –308 to1.7x10308 with a precision of 15 digits.
bool(Boolean)	can have only two possible values: true and false.
Void	Valueless

C++ allows certain of the basic types to have modifiers preceding them. A modifier alters the meaning of the base type so that it more precisely fits the needs of various situations. The data type modifiers are: signed, unsigned, long and short

Type	Minimal Range
char	–127 to 127
unsigned char	0 to 255
signed char	–127 to 127
int	–32,767 to 32,767
unsigned int	0 to 65,535
signed int	Same as **int**
short int	–32,767 to 32,767
unsigned short int	0 to 65,535
signed short int	Same as **short int**
long int	–2,147,483,647 to 2,147,483,647
signed long int	Same as **long int**
unsigned long int	0 to 4,294,967,295
float	1E–37 to 1E+37, with six digits of precision
double	1E–37 to 1E+37, with ten digits of precision
long double	1E–37 to 1E+37, with ten digits of precision

This figure shows all combinations of the basic data types and modifiers along with their size and range for a 16-bit word machine

Variable

A variable is a named area in memory used to store values during program execution. Variables are run time entities. A variable has a symbolic name and can be given a variety of values. When a variable is given a value, that value is actually placed in the memory space assigned to the variable. All variables must be declared before they can be used. The general form of a declaration is:

type variable_list;

Here, type must be a valid data type plus any modifiers, and variable_list may consist of one or more identifier names separated by commas. Here are some declarations:

int i,j,l;

short int si;

unsigned int ui;

double balance, profit, loss;

Constants

Constants refer to fixed values that the program cannot alter. Constants can be of any of the basic data types. The way each constant is represented depends upon its type. Constants are also called literals. We can use keyword const prefix to declare constants with a specific type as follows:

const type variableName = value;

e.g,

const int LENGTH = 10;

Enumerated Types

An enumerated type declares an optional type name and a set of zero or more identifiers that can be used as values of the type. Each enumerator is a constant whose type is the enumeration. Creating an enumeration requires the use of the keyword enum. The general form of an enumeration type is:

enum enum-name { list of names } var-list;

Here, the enum-name is the enumeration's type name. The list of names is comma separated.

For example, the following code defines an enumeration of colors called color and the variable c of type color. Finally, c is assigned the value "blue".

enum color { red, green, blue } =c;

By default, the value of the first name is 0, the second name has the value 1 and the third has the value 2, and so on. But you can give a name, a specific value by adding an initializer. For example, in the following enumeration, green will have the value 5.

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

Here, blue will have a value of 6 because each name will be one greater than the one that precedes it.

Operator

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. C++ is rich in built-in operators. Generally, there are six type of operators: Arithmetical operators, Relational operators, Logical operators, Assignment operators, Conditional operators, Comma operator.

Arithmetical operators

Arithmetical operators +, -, *, /, and % are used to performs an arithmetic (numeric) operation.

Operator	Meaning
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus

You can use the operators +, -, *, and / with both integral and floating-point data types. Modulus or remainder % operator is used only with the integral data type.

Relational operators

The relational operators are used to test the relation between two values. All relational operators are binary operators and therefore require two operands. A relational expression returns zero when the relation is false and a non-zero when it is true. The following table shows the relational operators.

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

Logical operators

The logical operators are used to combine one or more relational expression. The logical operators are

Operators	Meaning
||	OR
&&	AND
!	NOT

Assignment operator

The assignment operator '=' is used for assigning a variable to a value. This operator takes the expression on its right-hand-side and places it into the variable on its left-hand-side. For example:

m = 5;

The operator takes the expression on the right, 5, and stores it in the variable on the left, m.

x = y = z = 32;

This code stores the value 32 in each of the three variables x, y, and z. In addition to standard assignment operator shown above, C++ also support compound assignment operators.

Compound Assignment Operators

Operator	Example	Equivalent to
+ =	A + = 2	A = A + 2
- =	A - = 2	A = A – 2
% =	A % = 2	A = A % 2
/=	A/ = 2	A = A / 2
*=	A * = 2	A = A * 2

Increment and Decrement Operators

C++ provides two special operators viz '++' and '--' for incrementing and decrementing the value of a variable by 1. The increment/decrement operator can be used with any type of variable but it cannot be used with any constant. Increment and decrement operators each have two forms, pre and post.

The syntax of the increment operator is:
Pre-increment: ++variable
Post-increment: variable++
The syntax of the decrement operator is:

Pre-decrement: —variable
Post-decrement: variable—

In Prefix form first variable is first incremented/decremented, then evaluated
In Postfix form first variable is first evaluated, then incremented / decremented.

Conditional operator

The conditional operator ?: is called ternary operator as it requires three operands. The format of the conditional operator is :

Conditional_ expression ? expression1 : expression2;
If the value of conditional expression is true then the expression1 is evaluated, otherwise expression2 is evaluated.

int a = 5, b = 6;

big = (a > b) ? a : b;
The condition evaluates to false, therefore big gets the value from b and it becomes 6.

The comma operator

The comma operator gives left to right evaluation of expressions. When the set of expressions has to be evaluated for a value, only the rightmost expression is considered.

int a = 1, b = 2, c = 3, i; // comma acts as separator, not as an operator

i = (a, b); // stores b into i would first assign the value of a to i, and then assign value of b to variable i. So, at the end, variable i would contain the value 2.

The sizeof operator

The sizeof operator can be used to find how many bytes are required for an object to store in memory. For example

sizeof (char) returns 1

sizeof (float) returns 4

Typecasting

Typecasting is the concept of converting the value of one type into another type. For example, you might have a float that you need to use in a function that requires an integer.

Implicit conversion

Almost every compiler makes use of what is called automatic typecasting. It automatically converts one type into another type. If the compiler converts a type it will normally give a warning. For example this warning: conversion from ‘double’ to ‘int’, possible loss of data. The problem with this is, that you get a warning (normally you want to compile without warnings and errors) and you are not in control. With control we mean, you did not decide to convert to another type, the compiler did. Also the possible loss of data could be unwanted.

Explicit conversion

The C++ language have ways to give you back control. This can be done with what is called an explicit conversion.

Four typecast operators

The C++ language has four typecast operators:

- static_cast
- reinterpret_cast
- const_cast
- dynamic_cast

Type Conversion

The Type Conversion is that which automatically converts the one data type into another but remember we can store a large data type into the other. For example we can't store a float into int because a float is greater than int.

When a user can convert the one data type into then it is called as the **type casting.** The type Conversion is performed by the compiler but a casting is done by the user for example converting a float into int. When we use the Type Conversion then it is called the promotion. In the type casting when we convert a large data type into another then it is called as the demotion. When we use the type casting then we can loss some data.

Control Structures

Control structures allows to control the flow of program's execution based on certain conditions C++ supports following basic control structures:

1) Selection Control structure

2) Loop Control structure

1) Selection Control structure:

Selection Control structures allows to control the flow of program's execution depending upon the state of a particular condition being true or false .C++ supports two types of selection statements :if and switch. Condition operator (?:) can also be used as an alternative to the if statement.

A.1) If Statement:

The syntax of an if statement in C++ is:

if(condition)

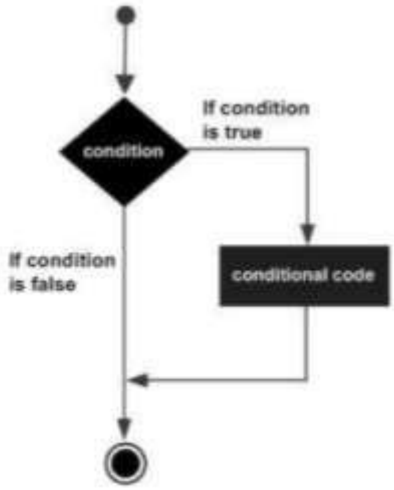
{

// statement(s) will execute if the condition is true

}

If the condition evaluates to true, then the block of code inside the if statement will be executed. If it evaluates to false, then the first set of code after the end of the if statement (after the closing curly brace) will be executed.

Flowchart showing working of if statement



// A Program to find whether a given number is even or odd using if … else statement

```
#include<iostream.h>
#include<conio.h>
int main()
{  int n;
cout<<"enter number";
cin>>n;
if(n%2==0)
cout<<"Even number";
else
cout<<"Odd number";
return 0;
}
```

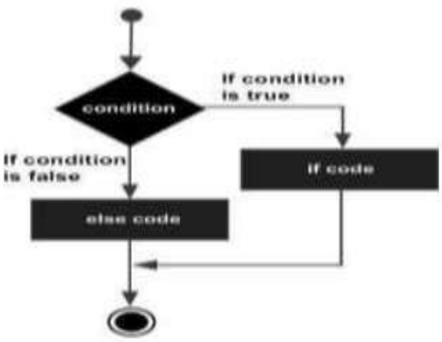
A.2) The if...else Statement

The syntax is shown as:

```
if(condition){
// statement(s) will execute if the condition is true
}
else{
// statement(s) will execute if condition is false
}
```

If the condition evaluates to true, then the if block of code will be executed, otherwise else block of code will be executed.

Flowchart



A.3) if...else if...else Statement

An if statement can be followed by an optional else if...else statement, which is very useful to test various conditions using single if...else if statement.

The Syntax is shown as:

```
if(condition 1){
// Executes when the condition 1 is true
}
else if(condition 2){
// Executes when the condition 2 is true
}
else if(condition 3){
// Executes when the condition 3 is true
}
else {
// executes when the none of the above condition is true.
}
```

A.4) Nested if Statement

It is always legal to nest if-else statements, which means you can use one if or

else if statement inside another if or else if statement(s).

The syntax for a nested if statement is as follows:

```
if( condition 1){
// Executes when the condition 1 is true
 if(condition 2){
// Executes when the condition 2 is true
         }
}
```

B) Switch

C++ has a built-in multiple-branch selection statement, called switch, which successively tests the value of an expression against a list of integer or character constants. When a match is found, the statements associated with that constant are executed. The general form of the switch statement is:

switch (expression) {

case constant1:

statement sequence

break;

case constant2:

statement sequence

break;

case constant3:

statement sequence

break;

.

.

default

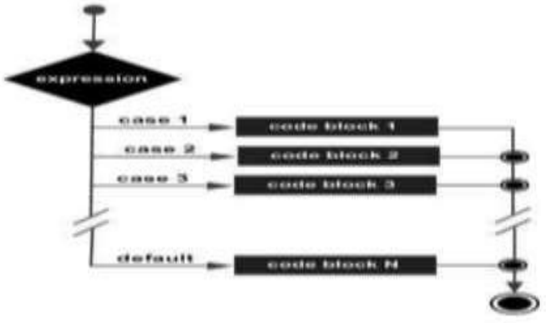
statement sequence

}

The expression must evaluate to a character or integer value. Floating-point expressions, for example, are not allowed. The value of expression is tested, in order, against the values of the constants specified in the case statements. When a match is found, the statement sequence associated with that case is executed until the break statement or the end of the switch statement is reached. The default statement is executed if no matches are found. The default is optional and, if it is not present, no action takes place if all matches fail.

The break statement is one of C++'s jump statements. You can use it in loops as well as in the switch statement. When break is encountered in a switch, program execution "jumps" to the line of code following the switch statement.

Flowchart



2) Loop control structures

A loop statement allows us to execute a statement or group of statements multiple times. Loops or iterative statements tell the program to repeat a fragment of code several times or as long as a certain condition holds. C++ provides three convenient iterative statements: while, for, and do-while.

while loop

A while loop statement repeatedly executes a target statement as long as a given condition is true. It is an entry-controlled loop.

The syntax of a while loop in C++ is:

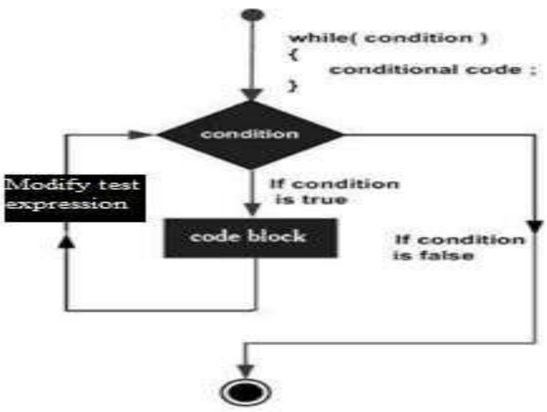
```
while(condition){

statement(s);

}
```

Here, statement(s) may be a single statement or a block of statements. The condition may be any expression, and true is any non-zero value. The loop iterates while the condition is true. After each execution of the loop, the value of test expression is changed. When the condition becomes false, program control passes to the line immediately following the loop.

Flowchart




```
// A program to display numbers from 1 to 100

#include<iostream.h>

#include<conio.h>

int main(){

  int i=1;

while(i<=100){

cout<<i ;

  i++;

}

return 0;

}
```

The do-while Loop

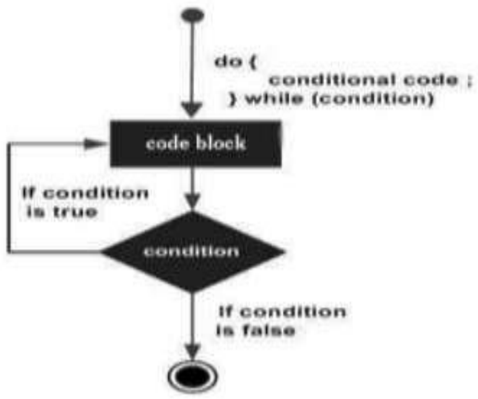
The do-while loop differs from the while loop in that the condition is tested after the body of the loop. This assures that the program goes through the iteration at least once. It is an exit-controlled loop.

```
do{

statement(s);

}while( condition );
```

The conditional expression appears at the end of the loop, so the statement(s) in the loop execute once before the condition is tested. If the condition is true, the flow of control jumps back up to do, and the statement(s) in the loop execute again. This process repeats until the given condition becomes false.




```
// A program to display numbers from 1 to 100

#include<iostream.h>

#include<conio.h>

int main( ){

 int i=1;

do

{           cout<<i ;

            i++;

} while(i<=100);

return 0;

}
```

for Loop

A for loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times. The syntax of a for loop in C++ is:

```
for ( init; condition; increment ){

statement(s);

}
```

Here is the flow of control in a for loop:

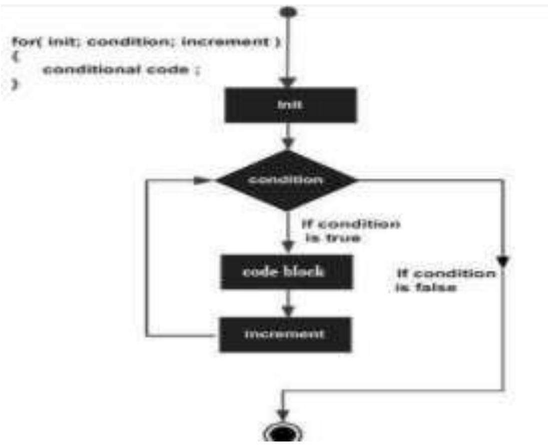
1. The init step is executed first, and only once. This step allows you to declare and initialize any loop control variables. You are not required to put a statement here, as long as a semicolon appears.

2. Next, the condition is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and flow of control jumps to the next statement just after the for loop.

3. After the body of the for loop executes, the flow of control jumps back up to the increment statement. This statement allows you to update any C++ loop control variables. This statement can be left blank, as long as a semicolon appears after the condition.

4. The condition is now evaluated again. If it is true, the loop executes and the process repeats itself (body of loop, then increment step, and then again condition). After the condition becomes false, the for loop terminates.

Flow Diagram




```
// A program to display numbers from 1 to 100

#include<iostream.h>

#include<conio.h>

int main() {

 int i ;

for (i=1;i<=100;i++)

{

cout<<i ;

return 0;

}
```

C++ Functions

A function groups a number of program statements into a unit and gives it a name. This unit can then be invoked from other parts of the program. The function's code is stored in only one place in memory, even though the function is executed many times in the course of the program's execution. Functions help to reduce the program size when same set of instructions are to be executed again and again. A general function consists of three parts, namely, function declaration (or prototype), function definition and function call.

Function declaration — prototype:

A function has to be declared before using it, in a manner similar to variables and constants. A function declaration tells the compiler about a function's name, return type, and parameters and how to call the function. The general form of a C++ function declaration is as follows:

return_type function_name(parameter list);

Function definition

The function definition is the actual body of the function. The function definition consists of two parts namely, function header and function body.

The general form of a C++ function definition is as follows:

return_type function_name(parameter list)

{ body of the function }

Here, **Return Type**: A function may return a value. The return_type is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword void.

Function Name: This is the actual name of the function.

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

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

Calling a Function

To use a function, you will have to call or invoke that function. To call a function, you simply need to pass the required parameters along with function name, and if function returns a value, then you can store returned value.

A c++ program calculating factorial of a number using functions

```
#include<iostream.h>

#include<conio.h>

int factorial(int n);                    //function declaration

int main(){
```

```
int no, f;
cout<<"enter the positive number:-";
cin>>no;
f=factorial(no);                //function call
cout<<"\nThe factorial of a number"<<no<<"is"<<f;
return 0;
}
int factorial(int n)            //function definition
{       int i , fact=1;
for(i=1;i<=n;i++){
        fact=fact*i;
}
return fact;
}
```

Inline Functions

An inline function is a function that is expanded inline at the point at which it is invoked, instead of actually being called. The reason that inline functions are an important addition to C++ is that they allow you to create very efficient code. Each time a normal function is called, a significant amount of overhead is generated by the calling and return mechanism. Typically, arguments are pushed onto the stack and various registers are saved when a function is called, and then restored when the function returns. The trouble is that these instructions take time. However, when a function is expanded inline, none of those operations occur. Although expanding function calls in line can produce faster run times, it can also result in larger code size because of duplicated code. For this reason, it is best to inline only very small functions. inline is actually just a request, not a command, to the compiler. The compiler can choose to ignore it. Also, some compilers may not inline all types of functions. If a function cannot be inlined, it will simply be called as a normal function.

A function can be defined as an inline function by prefixing the keyword inline to the function header as given below:

```
inline function header {
function body
}
```

```
// A program illustrating inline function
#include<iostream.h>
#include<conio.h>
inline int max(int x, int y){
if(x>y)
return x;
```

```
else
return y;
}
int main( ) {
int a,b;
cout<<"enter two numbers";
cin>>a>>b;
cout << "The max is: " <<max(a,,b) << endl;
return 0;
}
```

Macros Vs inline functions

Preprocessor macros are just substitution patterns applied to your code. They can be used almost anywhere in your code because they are replaced with their expansions before any compilation starts. Inline functions are actual functions whose body is directly injected into their call site. They can only be used where a function call is appropriate.

inline functions are similar to macros (because the function code is expanded at the point of the call at compile time), inline functions are parsed by the compiler, whereas macros are expanded by the preprocessor. As a result, there are several important differences:

- Inline functions follow all the protocols of type safety enforced on normal functions.
- Inline functions are specified using the same syntax as any other function except that they include the inline keyword in the function declaration.
- Expressions passed as arguments to inline functions are evaluated once.
- In some cases, expressions passed as arguments to macros can be evaluated more than once.
- macros are expanded at pre-compile time, you cannot use them for debugging, but you can use inline functions.

Reference variable

A reference variable is an alias, that is, another name for an already existing variable. Once a reference is initialized with a variable, either the variable name or the reference name may be used to refer to the variable. To declare a reference variable or parameter, precede the variable's name with the &.The syntax for declaring a reference variable is:

datatype &Ref = variable name;

Example:

```
int main(){
int var1=10;                    //declaring simple variable
int & var2=var1; //declaring reference variable
cout<<"\n value of var2 =" << var2;
```

```
return 0;

}
```

var2 is a reference variable to var1.Hence, var2 is an alternate name to var1.This code prints the value of var2 exactly as that of var1.

Call by reference

Arguments can be passed to functions in one of two ways: using call-by-value or call-by-reference. When using call-by-value, a copy of the argument is passed to the function. Call-by-reference passes the address of the argument to the function. By default, C++ uses call-by-value.

Provision of the reference variables in c++ permits us to pass parameter to the functions by reference. When we pass arguments by reference, the formal arguments in the called function become aliases to the actual arguments in the calling function. This means that when the function is working with its own arguments, it is actually working on the original data.

Example

```
#include <iostream.h>

#include<conio.h>

void swap(int &x, int &y); // function declaration

int main (){

int a = 10, b=20;

cout << "Before swapping”<<endl;

cout<< “value of a :" << a <<” value of b :" << b << endl;

swap(a, b);          //calling a function to swap the values.

cout << "After swapping”<<endl;

cout<<” value of a :" << a<< “value of b :" << b << endl;

return 0;

}

void swap(int &x, int &y)  {          //function definition to swap the values.

int temp;

temp = x;

x = y;

y = temp;

}
```

Output:

Before swapping   value of a:10 value of b:20

After swapping   value of a:20 value of b:10

Unit-II

Class

A class is a user defined data type. A class is a logical abstraction. It is a template that defines the form of an object. A class specifies both code and data. It is not until an object of that class has been created that a physical representation of that class exists in memory. When you define a class, you declare the data that it contains and the code that operates on that data. Data is contained in instance variables defined by the class known as data members, and code is contained in functions known as member functions. The code and data that constitute a class are called members of the class. The general form of class declaration is:

```
class class-name {

access-specifier:

data and functions

access-specifier:

data and functions

// ...

access-specifier:

data and functions

} object-list;
```

The object-list is optional. If present, it declares objects of the class. Here, access-specifier is one of these three C++ keywords:

public   private   protected

By default, functions and data declared within a class are private to that class and may be accessed only by other members of the class. The public access_specifier allows functions or data to be accessible to other parts of your program. The protected access_specifier is needed only when inheritance is involved.

Example:

```
#include<iostream.h>

#include<conio.h>

Class myclass {            // class declaration

// private members to myclass

int a;

public:

// public members to myclass

void set_a(intnum);

int get_a( );

};
```

Object

An object is an identifiable entity with specific characteristics and behavior. An object is said to be an instance of a class. Defining an object is similar to defining a variable of any data type: Space is set aside for it in memory. Defining objects in this way means creating them. This is also called instantiating them. Once a Class has been declared, we can create objects of that Class by using the class Name like any other built-in type variable as shown:

className objectName

Example

```
void main( ) {
myclass ob1, ob2;                    //these are object of type myclass
// ... program code
}
```

Accessing Class Members

The main() cannot contain statements that access class members directly. Class members can be accessed only by an object of that class. To access class members, use the dot (.) operator. The dot operator links the name of an object with the name of a member. The general form of the dot operator is shown here:

object.member

Example

ob1.set_a(10);

The private members of a class cannot be accessed directly using the dot operator, but through the member functions of that class providing data hiding. A member function can call another member function directly, without using the dot operator.

C++ program to find sum of two numbers using classes

```
#include<iostream.h>
#include<conio.h>
class A{
 int a,b,c;
 public:
 void sum(){
 cout<<"enter two numbers";
 cin>>a>>b;
 c=a+b;
 cout<<"sum="<<c;
 }
};
```

```
int main(){
 A u;
 u.sum();
 getch();
 return(0);
}
```

Scope Resolution operator

Member functions can be defined within the class definition or separately using scope resolution operator (::). Defining a member function within the class definition declares the function inline, even if you do not use the inline specifier. Defining a member function using scope resolution operator uses following declaration

```
return-type class-name::func-name(parameter- list) {
// body of function
}
```

Here the class-name is the name of the class to which the function belongs. The scope resolution operator (::) tells the compiler that the function func-name belongs to the class class-name. That is, the scope of the function is restricted to the class-name specified.

```
Class myclass {
int a;
public:
void set_a(intnum); //member function declaration
int get_a( );                    //member function declaration
};
//member function definition outside class using scope resolution operator
void myclass :: set_a(intnum)
{
a=num;
}
int myclass::get_a( ) {
return a;
}
```

Another use of scope resolution operator is to allow access to the global version of a variable. In many situation, it happens that the name of global variable and the name of the local variable are same .In this while accessing the variable, the priority is given to the local variable by the compiler. If we want to access or use the global variable, then the scope resolution operator (::) is used. The syntax for accessing a global variable using scope resolution operator is as follows:-

:: Global-variable-name

Static Data Members

When you precede a member variable's declaration with static, you are telling the compiler that only one copy of that variable will exist and that all objects of the class will share that variable. Unlike regular data members, individual copies of a static member variable are not made for each object. No matter how many objects of a class are created, only one copy of a static data member exists. Thus, all objects of that class use that same variable. All static variables are initialized to zero before the first object is created. When you declare a static data member within a class, you are not defining it. (That is, you are not allocating storage for it.) Instead, you must provide a global definition for it elsewhere, outside the class. This is done by redeclaring the static variable using the scope resolution operator to identify the class to which it belongs. This causes storage for the variable to be allocated.

One use of a static member variable is to provide access control to some shared resource used by all objects of a class. Another interesting use of a static member variable is to keep track of the number of objects of a particular class type that are in existence.

Static Member Functions

Member functions may also be declared as static. They may only directly refer to other static members of the class. Actually, static member functions have limited applications, but one good use for them is to "preinitialize" private static data before any object is actually created. A static member function can be called using the class name instead of its objects as follows:

class name :: function name

```
//Program showing working of static class members

#include <iostream.h>

#include<conio.h>

class static_type {

static int i;                                   //static data member

public:

static void init(int x) {i = x;}          //static member function

void show() {cout << i;}};

int static_type :: i;                           // static data member definition

int main(){

static_type::init(100);                  //Accessing static function

static_type x;

x.show();

return 0;

}
```

Constructor:

A constructor is a special member function whose task is to initialize the objects of its class. It is special because its name is same as the class name. The constructor is invoked whenever an object of its associated class is created. It is called constructor because it construct the value data members of the class. The constructor functions have some special characteristics.

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

Example:

```
#include< iostream.h>

#include<conio.h>

class myclass {                // class declaration

int a;

public:

myclass( );                    //default constructor

void show( );

};

myclass :: myclass( ) {

cout <<"In constructor\n";

a=10;

}

myclass :: show( ) {

cout<< a;

}

int main( ) {

int ob; // automatic call to constructor

ob.show( );

return0;

}
```

In this simple example the constructor is called when the object is created, and the constructor initializes the private variable a to10.

Default constructor

The default constructor for any class is the constructor with no arguments. When no arguments are passed, the constructor will assign the values specifically assigned in the body of the constructor. It can be zero or any other value. The default constructor can be identified by the name of the class followed by empty parentheses. Above program uses default constructor. If it is not defined explicitly, then it is automatically defined implicitly by the system.

Parameterized Constructor

It is possible to pass arguments to constructors. Typically, these arguments help initialize an object when it is created. To create a parameterized constructor, simply add parameters to it the way you would to any other function. When you define the constructor's body, use the parameters to initialize the object.

```
#include <iostream.h>

#include<conio.h>

class myclass {

int a, b;

public:

myclass(int i, int j) //parameterized constructor

 {a=i; b=j;}

void show() { cout << a << " " << b;}

};

int main() {

myclass ob(3, 5); //call to constructor

ob.show();

return 0;

}
```

C++ supports constructor overloading. A constructor is said to be overloaded when the same constructor with different number of argument and types of arguments initializes an object.

Copy Constructors

The copy constructor is a constructor which creates an object by initializing it with an object of the same class, which has been created previously. If class definition does not explicitly include copy constructor, then the system automatically creates one by default. The copy constructor is used to:

- Initialize one object from another of the same type.
- Copy an object to pass it as an argument to a function.
- Copy an object to return it from a function.

The most common form of copy constructor is shown here:

```
classname (const classname &obj) {

// body of constructor
```

}

Here, obj is a reference to an object that is being used to initialize another object. The keyword const is used because obj should not be changed.

Destructor

A destructor destroys an object after it is no longer in use. The destructor, like constructor, is a member function with the same name as the class name. But it will be preceded by the character Tilde (~).A destructor takes no arguments and has no return value. Each class has exactly one destructor. . If class definition does not explicitly include destructor, then the system automatically creates one by default. It will be invoked implicitly by the compiler upon exit from the program to clean up storage that is no longer accessible.

```
// A Program showing working of constructor and destructor

#include<iostream.h>

#include<conio.h>

class Myclass{

public:

int x;

Myclass(){                    //Constructor

x=10; }

~Myclass(){                   //Destructor

cout<<"Destructing…." ;

}

int main(){

Myclass ob1, ob2;

cout<<ob1.x<<" "<<ob2.x;

return 0; }
```

Output:

10 10

Destructing……..

Destructing……..

Friend function

In general, only other members of a class have access to the private members of the class. However, it is possible to allow a nonmember function access to the private members of a class by declaring it as a friend of the class. To make a function a friend of a class, you include its prototype in the class declaration and precede it with the friend keyword. The function is declared with friend keyword. But while defining friend function, it does not use either keyword friend or :: operator. A function can be a friend of more than one class. Member function of one class can be friend functions of another class. In such cases they are defined using the scope resolution operator.

A friend, function has following characteristics.

- It is not in the scope of the class to which it has been declared as friend.
- A friend function cannot be called using the object of that class. If can be invoked like a normal function without help of any object.
- It cannot access the member variables directly & has to use an object name dot membership operator with member name.
- It can be declared either in the public or the private part of a class without affecting its meaning.
- Usually, it has the object as arguments.

Program to illustrate use of friend function

```
#include<iostream.h>
#include<conio.h>
class A{
  int x, y;
 public:
  friend void display(A &obj);
  void  getdata()  {
  cin>>x>>y;
  }
};
void display(A &obj){
 cout<<obj.x<<obj.y;
}
int main(){
 A a;
 a.getdata();
 display(a);
 getch();
 return 0;
}
```

Operator overloading

There is another useful methodology in C++ called operator overloading. The language allows not only functions to be overloaded, but also most of the operators, such as +, -, *, /, etc. As the name suggests, here the conventional operators can be programmed to carry out more complex operations. This overloading concept is fundamentally the same i.e. the same operators can be made to perform different operations depending on the context. Such operators have to be specifically defined and appropriate function programmed. When an operator is overloaded, none of its original meaning is lost. It is simply that a new operation, relative to a specific class, is defined. For example, a class that defines a linked list might use the + operator to add an object to the list. A class that implements a stack might use the + to push an object onto the stack.

An operator function defines the operations that the overloaded operator will perform relative to the class upon which it will work. An operator function is created using the keyword operator. The general form of an operator function is

```
type classname::operator#(arg-list) { // operations

}
```

Here, the operator that you are overloading is substituted for the #, and type is the type of value returned by the specified operation. Operator functions can be either members or nonmembers of a class. Nonmember operator functions are often friend functions of the class.

These operators cannot be overloaded:- ., : :, .*, ?

The process of overloading involves the following steps:

- Create a class that defines the data type that is to be used in the overloading operation.
- Declare the operator function operator op() in the public part of the class.
- Define the operator function to implement the required operations.

Overloading a unary operator using member function

Overloading a unary operator using a member function, the function takes no parameters. Since, there is only one operand, it is this operand that generates the call to the operator function. There is no need for another parameter.

Overloading unary minus operator

```
#include<iostream.h>

#include<conio.h>

class A {

 int x,y,z;

 public:

  void getdata(int a,int b,int c) {

   x=a;

   y=b;

   z=c;

  }

  void display() {

   cout<<"\nx="<<x<<"\ny="<<y<<"\nz="<<z;
```

```
 }
 void operator -() //unary minus overload function
 {
  x=-x;
  y=-y;
  z=-z;
 }
};
int main() {
 A a;
 a.getdata(2,3,4);
 a.display();
 -a;                       //activates operator –() function
 a.display();
 getch();
 return 0;
}
```

Overloading binary operator

When a member operator function overloads a binary operator, the function will have only one parameter. This parameter will receive the object that is on the right side of the operator. The object on the left side is the object that generates the call to the operator function and is passed implicitly by this pointer. ‘this’ can be used in overloading + operator .

```
#include<iostream.h>
#include<conio.h>
class A{
 int x,y;
 public:
 void input() {
  cin>>x>>y;
 }
 void display() {
 cout<<"\nx="<<x<<"\ny="<<y<<"\nx+y="<<x+y;
 }
 A operator+(A p );          //overload binary + operator
};
```

```
A A :: operator+(A p) {
  A t;
  t.x=x + p.x;
  t.y=y + p.y;
  return t;
}
int main(){
A a1, a2, a3;
a1.input();
a2.input();
a3=a2+a1;                    //activates operator+() function
a3.display();
getch();
return 0;
}
```

this Pointer

It is facilitated by another interesting concept of C++ called this pointer. ‘this’ is a C++ keyword. ‘this’ always refers to an object that has called the member function currently. We can say that ‘this’ is a pointer. It points to the object that has called this function this time. While overloading binary operators, we use two objects, one that called the operator function and the other, which is passed to the function. We referred to the data member of the calling object, without any prefix. However, the data member of the other object had a prefix. Always ‘this’ refers to the calling object place of the object name.

Function Overloading

Function overloading is the process of using the same name for two or more functions. Each redefinition of the function must use either different types of parameters or a different number of parameters. It is only through these differences that the compiler knows which function to call in any given situation. Overloaded functions can help reduce the complexity of a program by allowing related operations to be referred to by the same name. To overload a function, simply declare and define all required versions. The compiler will automatically select the correct version based upon the number and/or type of the arguments used to call the function. Two functions differing only in their return types cannot be overloaded.

Example

```
#include<iostream.h>
#include<conio.h>
int sum(int p,int q,int r);
double sum(int l,double m);
float sum(float p,float q)
int main(){
```

```
cout<<"sum="<< sum(11,22,33);           //calls  func1
cout<<"sum="<< sum(10,15.5);            //calls  func2
cout<<"sum="<< sum(13.5,12.5);          //calls  func3
return 0;
}
int sum(int p,int q,int r){             //func1
  return(a+b+c);
 }
 double sum(int l,double m){            //func2
   return(l+m);
 }
  float sum(float p,float q){           //func3
  return(p+q);
 }
```

Default arguments

C++ allows a function to assign a parameter a default value when no argument corresponding to that parameter is specified in a call to that function. The default value is specified in a manner syntactically similar to a variable initialization. All default parameters must be to the right of any parameters that don't have defaults. We cannot provide a default value to a particular argument in the middle of an argument list. When you create a function that has one or more default arguments, those arguments must be specified only once: either in the function's prototype or in the function's definition if the definition precedes the function's first use.

Default arguments are useful if you don’t want to go to the trouble of writing arguments that, for example, almost always have the same value. They are also useful in cases where, after a program is written, the programmer decides to increase the capability of a function by adding another argument. Using default arguments means that the existing function calls can continue to use the old number of arguments, while new function calls can use more.

Example

```
#include <iostream.h>

#include<conio.h>

int sum(int a, int b=20){

return( a + b);

}

int main (){

int a = 100, b=200, result;

result = sum(a, b);          //here a=100 , b=200

cout << "Total value is :" << result << endl;

result = sum(a);             //here a=100 , b=20(using default value)

cout << "Total value is :" << result << endl;

return 0;

}
```

Unit-III

Inheritance

Inheritance is the mechanism by which one class can inherit the properties of another. It allows a hierarchy of classes to be build, moving from the most general to the most specific. When one class is inherited by another, the class that is inherited is called the base class. The inheriting class is called the derived class. In general, the process of inheritance begins with the definition of a base class. The base class defines all qualities that will be common to any derived class. In essence, the base class represent the most general description of a set of traits. The derived class inherits those general traits and adds properties that are specific to that class. When one class inherits another, it uses this general form:

```
class derived-class-name : access base-class-name{

// ...

}
```

Here access is one of the three keywords: public, private, or protected. The access specifier determines how elements of the base class are inherited by the derived class.

When the access specifier for the inherited base class is **public**, all public members of the base class become public members of the derived class. If the access specifier is **private**, all public members of the base class become private members of the derived class. In either case, any private members of the base class remain private to it and are inaccessible by the derived class.

It is important to understand that if the access specifier is **private**, public members of the base become private members of the derived class. If the access specifier is not present, it is private by default.

The **protected** access specifier is equivalent to the private specifier with the sole exception that protected members of a base class are accessible to members of any class derived from that base. Outside the base or derived classes, protected members are not accessible. When a protected member of a base class is inherited as public by the derived class, it becomes a protected member of the derived class. If the base class is inherited as private, a protected member of the base becomes a private member of the derived class. A base class can also be inherited as protected by a derived class. When this is the case, public and protected members of the base class become protected members of the derived class (of course, private members of the base remain private to it and are not accessible by the derived class).

Program to illustrate concept of inheritance

```
#include<iostream.h>

#include<conio.h>

class base                          //base class

{

 int x,y;

 public:

  void show() {

   cout<<"In base class";

  }

};
```

```
class derived : public base               //derived class
{
 int a,b;
 public:
  void show2()  {
   cout<<"\nIn derived class";
  }
};
 int main() {
  derived d;
  d.show();          //uses base class's show() function
  d.show2();         //uses derived class's show2() function
  getch();
  return 0;
 }
```

Types of Inheritances

Single Inheritance

The process in which a derived class inherits traits from only one base class, is called single inheritance. In single inheritance, there is only one base class and one derived class. The derived class inherits the behavior and attributes of the base class. However the vice versa is not true. The derived class can add its own properties i.e. data members (variables) and functions. It can extend or use properties of the base class without any modification to the base class. We declare the base class and derived class as given below:

```
class base_class {
};
class derived_ class : visibility-mode base_ class {
};
```

Program to illustrate concept of single inheritance

```
#include<iostream.h>
#include<conio.h>
class base                        //base class
{
 int x,y;
 public:
  void show() {
   cout<<"In base class";
  }
```

```
};
class derived : public base //derived class
{
 int a,b;
 public:
  void show2()  {
   cout<<"\nIn derived class";
  }
};
 int main() {
  derived d;
  d.show();          //uses base class's show() function
  d.show2();         //uses derived class's show2() function
  getch();
  return 0;
 }
```

Ambiguity in single Inheritance

Whenever a data member and member functions are defined with the same name in both the base and derived class, ambiguity occurs. The scope resolution operator must be used to refer to particular class as: object name.class name :: class member

Multiple Inheritance

The process in which a derived class inherits traits from several base classes, is called multiple inheritance. In Multiple inheritance, there is only one derived class and several base classes. We declare the base classes and derived class as given below:

```
class base_class1{
};
class base_class2{
};
class derived_ class : visibility-mode base_ class1 , visibility-mode base_ class2 {
};
```

Multilevel Inheritance

The process in which a derived class inherits traits from another derived class, is called Multilevel Inheritance. A derived class with multilevel inheritance is declared as :

```
class base_class {
};
class derived_ class1 : visibility-mode base_ class {
```

};

class derived_ class 2: visibility-mode derived_ class1 {

};

Here, derived_ class 2 inherits traits from derived_ class 1 which itself inherits from base_class.

Hierarchical Inheritance

The process in which traits of one class can be inherited by more than one class is known as Hierarchical inheritance. The base class will include all the features that are common to the derived classes. A derived class can serve as a base class for lower level classes and so on.

Hybrid Inheritance

The inheritance hierarchy that reflects any legal combination of other types of inheritance is known as hybrid Inheritance.

Overriding

Overriding is defined as the ability to change the definition of an inherited method or attribute in a derived class. When multiple functions of the same name exist with different signatures it is called function overloading. When the signatures are the same, they are called function overriding. Function overriding allows a derived class to provide specific implementation of a function that is already provided by a base class. The implementation in the derived class overrides or replaces the implementation in the corresponding base class.

Virtual base classes

A potential problem exists when multiple base classes are directly inherited by a derived class. To understand what this problem is, consider the following class hierarchy:

Here the base class Base is inherited by both Derived1and Derived2. Derived3 directly inherits both Derived1and Derived2.However, this implies that Base is actually inherited twice by Derived3.First it is inherited through Derived1, and then again through Derived2.This causes ambiguity when a member of Base is used by Derived3. Since two copies of Base are included in Derived3, is a reference to a member of Base referring to the Base inherited indirectly through Derived1or to the Base inherited indirectly through Derived2? To resolve this ambiguity, C++ includes a mechanism by which only one copy of Base will be included in Derived3. This feature is called a virtual base class.

In situations like this, in which a derived class indirectly inherits the same base class more than once, it is possible to prevent multiple copies of the base from being present in the derived class by having that base class inherited as virtual by any derived classes. Doing this prevents two or more copies of the base from being present in any subsequent derived class that inherits the base class indirectly. The virtual keyword precedes the base class access specifier when it is inherited by a derived class.

```
// This program uses a virtual base class.

#include <iostream>

using namespace std;

class Base {

public:

int i;
```

```
};
// Inherit Base as virtual
classDerived1 : virtual publicBase {
public:
int j;
};
// Inherit Base as virtual here, too
classDerived2 : virtual publicBase {
public:
int k;
};
// Here Derived3 inherits both Derived1 and Derived2.
// However, only one copy of base is inherited.
class Derived3 : publicDerived1, publicDerived2 {
public:
int product( ) { return i*j*k; }
};
int main( ) {
Derived3 ob;
ob.i = 10; // unambiguous because virtual Base
ob.j = 3;
ob.k = 5;
cout << "Product is: " << ob.product( ) << "\n";
return 0;
}
```

If Derived1and Derived2 had not inherited Base as virtual, the statement ob.i=10 would have been ambiguous and a compile-time error would have resulted. It is important to understand that when a base class is inherited as virtual by a derived class, that base class still exists within that derived class.

For example, assuming the preceding program, this fragment is perfectly valid:

```
Derived1 ob;
ob.i = 100;
```

Friend Classes

It is possible for one class to be a friend of another class. When this is the case, the friend class and all of its member functions have access to the private members defined within the other class. For example,

```
// Using a friend class.
#include<iostream.h>
#include<conio.h>
class A{
  int x, y;
 public:
  friend void display(A &obj);
  friend class B;
  void getdata() {
   cin>>x>>y;
  }
};
class B{
 int  p,q;
 public:
 void get(A &obj) {
   p=obj.x;
   q=obj.y;
 }
};
void display(A &obj){
 cout<<obj.x<<obj.y;
}
int main(){
 A a;
 B b;
 b.get(a);
 a.g etdata();
 display(a);
 getch();
```

```
 return 0;

}
```

It is critical to understand that when one class is a friend of another, it only has access to names defined within the other class. It does not inherit the other class. Specifically, the members of the first class do not become members of the friend class.

Pointer

A pointer is a variable that contains a memory address. Very often this address is the location of another object, such as a variable. For example, if x contains the address of y, then x is said to "point to" y. Pointer variables must be declared as such. The general form of a pointer variable declaration is

type *var-name;

Here, type is the pointer's base type. The base type determines what type of data the pointer will be pointing to. var-name is the name of the pointer variable.

To use pointer:

- We define a pointer variable.
- Assign the address of a variable to a pointer.
- Finally access the value at the address available in the pointer variable. This is done by using unary operator * that returns the value of the variable located at the address specified by its operand.

  Example:

  int a=10; //normal variable

  int*p; //declare pointer

  p = &a; // Assign the address of a variable "a" to a pointer "p"

  cout<<"a="<<*p; //prints a=10

OBJECT POINTERS

We have been accessing members of an object by using the dot operator. However, it is also possible to access a member of an object via a pointer to that object. When a pointer is used, the arrow operator (->) rather than the dot operator is employed. We can declare an object pointer just as a pointer to any other type of variable is declared. Specify its class name, and then precede the variable name with an asterisk. To obtain the address of an object, precede the object with the & operator, just as you do when taking the address of any other type of variable.

Here is a simple example,

```
#include< iostream>

#include<conio.h>

class myclass {

int a;

public:

myclass(int x); //constructor

int get( );

};

myclass :: myclass(int x) {
```

```
a=x;
}
int myclass :: get( ) {
return a;
}
int main( ) {
myclass ob(120); //create object
myclass *p; //create pointer to object
p=&ob; //put address of ob into p
cout <<"value using object: " <<ob.get( );
cout <<"\n";
cout <<"value using pointer: " <<p->get( );
return0;
}
```

Notice how the declaration : myclass *p; creates a pointer to an object of myclass. It is important to understand that creation of an object pointer does not create an object. It creates just a pointer to one. The address of ob is put into p by using the statement:

p=&ob;

Finally, the program shows how the members of an object can be accessed through a pointer.

Pointers to Derived Types

Pointers to base classes and derived classes are related in ways that other types of pointers are not. In general, a pointer of one type cannot point to an object of another type. However, base class pointers and derived objects are the exceptions to this rule. In C++, a base class pointer can also be used to point to an object of any class derived from that base. For example, assume that you have a base class called B and a class called D, which is derived from B. Any pointer declared as a pointer to B can also be used to point to an object of type D. Therefore, given

```
B *p;        //pointer p to object of type B
B  B_ob;     //object of type B
D D_ob;      //object of type D
```

both of the following statements are perfectly valid:

```
p = &B_ob;   //p points to object B
p = &D_ob;   //p points to object D, which is an object derived from B
```

A base pointer can be used to access only those parts of a derived object that were inherited from the base class. Thus, in this example, p can be used to access all elements of D_ob inherited from B_ob. However, elements specific to D_ob cannot be accessed through p.

Another point to understand is that although a base pointer can be used to point to a derived object, the reverse is not true. That is, you cannot access an object of the base type by using a derived class pointer.

C++ Virtual Function

A virtual function is a member function that is declared within a base class and redefined by a derived class. In order to make a function virtual, you have to add keyword virtual in front of a function definition. When a class containing a virtual function is inherited, the derived class redefines the virtual function relative to the derived class. The virtual function within the base class defines the form of the interface to that function. Each redefinition of the virtual function by a derived class implements its operation as it relates specifically to the derived class. That is, the redefinition creates a specific method. When a virtual function is redefined by a derived class, the keyword virtual is not needed. A virtual function can be called just like any member function. However, what makes a virtual function interesting, and capable of supporting run-time polymorphism, is what happens when a virtual function is called through a pointer. When a base pointer points to a derived object that contains a virtual function and that virtual function is called through that pointer, C++ determines which version of that function will be executed based upon the type of object being pointed to by the pointer. And this determination is made at run time. Therefore, if two or more different classes are derived from a base class that contains a virtual function, then when different objects are pointed to by a base pointer, different versions of the virtual function are executed.

```
// A simple example using a virtual function.

#include<iostream.h>

#include<conio.h>

class base {

public:

virtual void func( ) {

cout<< "Using base version of func(): ";

}

};

class derived1 : public base {

public:

voidfunc( ) {

cout<< "Using derived1's version of func(): ";

}

};

class derived2 : public base {

public:

voidfunc( ) {
```

```
cout<< "Using derived2's version of func(): ";
}
};
int main( ) {
base *p;
base ob;
derived1 d_ob1;
derived2 d_ob2;
p = &ob;
p- >func( ); // use base's func( )
p = &d_ob1;
p- >func( ); // use derived1's func( )
p = &d_ob2;
p- >func( ); // use derived2's func( )
return 0;
}
```

Pure virtual functions

Sometimes when a virtual function is declared in the base class, there is no meaningful operation for it to perform. This situation is common because often a base class does not define a complete class by itself. Instead, it simply supplies a core set of member functions and variables to which the derived class supplies the remainder. When there is no meaningful action for a base class virtual function to perform, the implication is that any derived class must override this function. To ensure that this will occur, C++ supports pure virtual functions. A pure virtual function has no definition relative to the base class. Only the function prototype is included. To make a pure virtual function, use this general form:

virtual type func-name(parameter-list) = 0;

The key part of this declaration is the setting of the function equal to 0. This tells the compiler that no body exists for this function relative to the base class. When a virtual function is made pure, it forces any derived class to override it. If a derived class does not, a compile-time error results. Thus, making a virtual function pure is a way to guaranty that a derived class will provide its own redefinition.

Abstract class

When a class contains atleast one pure virtual function, it is referred to as an abstract class. Since, an abstract class contains atleast one function for which no body exists, it is, technically, an incomplete type, and no objects of that class can be created. Thus, abstract classes exist only to be inherited. It is important to understand, however, that you can still create a pointer to an abstract class, since it is through the use of base class pointers that run-time polymorphism is achieved. (It is also possible to have a reference to an abstract class.) .

Unit-IV

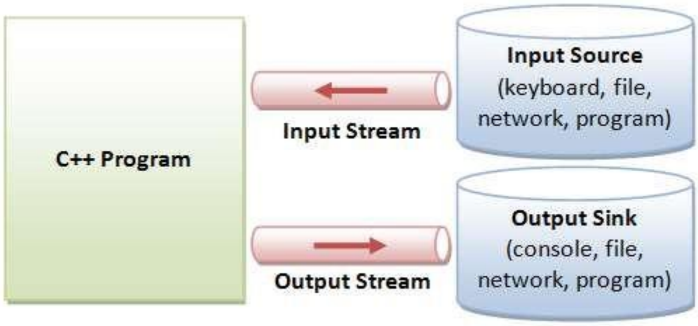
C++ Streams

The C++ I/O system operates through streams. A stream is logical device that either produces or consumes information. A stream is linked to a physical device by the C++ I/O system. All streams behave in the same manner, even if the actual physical devices they are linked to differ. Because all streams act the same, the I/O system presents the programmer with a consistent interface.

Two types of streams:

Output stream: a stream that takes data from the program and sends (writes) it to destination.

Input stream: a stream that extracts (reads) data from the source and sends it to the program.



C++ provides both the formatted and unformatted IO functions. In formatted or high-level IO, bytes are grouped and converted to types such as int, double, string or user-defined types. In unformatted or low-level IO, bytes are treated as raw bytes and unconverted. Formatted IO operations are supported via overloading the stream insertion (<<) and stream extraction (>>) operators, which presents a consistent public IO interface.

C++ provides supports for its I/O system in the header file< iostream>. Just as there are different kinds of I/O (for example, input, output, and file access), there are different classes depending on the type of I/O. The following are the most important stream classes:

Class istream :- Defines input streams that can be used to carry out formatted and unformatted input operations. It contains the overloaded extraction (>>) operator functions. Declares input functions such get(), getline() and read().

Class ostream :- Defines output streams that can be used to write data. Declares output functions put and write().The ostream class contains the overloaded insertion (<<) operator function

When a C++ program begins, these four streams are automatically opened:

Stream	**Meaning**	**Default Device**
cin	Standard input	Keyboard
cout	Standard output	Screen
cerr	Standard error	Screen
clog	Buffer version of cerr	Screen

Cin and Cout objects

cout is an object of class ostream. The cout is a predefined object that represents the standard output stream in C++. Here, the standard output stream represents monitor. In the language of C++, the << operator is referred to as the insertion operator because it inserts data into a stream. It inserts or sends the contents of variable on its right to the object on its left.

For example:

cout << ``Programming in C++";

Here << operator is called the stream insertion operator and is used to push data into a stream (in this case the standard output stream)

cin is an object of class istream. cin is a predefined object that corresponds to the standard input stream. The standard input stream represents keyboard. The >> operator is called the extraction operator because it extracts data from a stream. It extracts or takes the value from the keyboard and assigns it to the variable on it's right.

For example:

int number;

cin >> number;

Here >> operator accepts value from keyboard and stores in variable number.

Unformatted Input/Output Functions

Functions get and put

The get function receives one character at a time. There are two prototypes available in C++ for get as given below:

get (char *)

get ()

Their usage will be clear from the example below:

char ch ;

cin.get (ch);

In the above, a single character typed on the keyboard will be received and stored in the character variable ch.

Let us now implement the get function using the other prototype:

char ch ;

ch = cin.get();

This is the difference in usage of the two prototypes of get functions.

The complement of get function for output is the put function of the ostream class. It also has two forms as given below:

cout.put (var);

Here the value of the variable var will be displayed in the console monitor. We can also display a specific character directly as given below:

cout.put (‘a’);

getline and write functions

C++ supports functions to read and write a line at one go. The getline() function will read one line at a time. The end of the line is recognized by a new line character, which is generated by pressing the Enter key. We can also specify the size of the line.

The prototype of the getline function is given below:

cin.getline (var, size);

When we invoke the above statement, the system will read a line of characters contained in variable var one at a time. The reading will stop when it encounters a new line character or when the required number (size-1) of characters have been read, whichever occurs earlier. The new line character will be received when we enter a line of size less than specified and press the Enter key. The Enter key or Return key generates a new line character. This character will be read by the function but converted into a NULL character and appended to the line of characters.

Similarly, the write function displays a line of given size. The prototype of the write function is given below:

write (var, size) ;

where var is the name of the string and size is an integer.

Formatted I/O via manipulators

The C++ I/O system allows you to format I/O operations. For example, you can set a field width, specify a number base, or determine how many digits after the decimal point will be displayed. I/O manipulators are special I/O format functions that can occur within an I/O statement.

Manipulator	Purpose	Input/Ouput
boolalpha	Turns on boolaphaflag	Input/Output
dec	Turns on decflag	Input/Output
endl	Outputs a newline character and flushes the stream	Output
ends	Outputs a null	Output
fixed	Turns on fixed flag	Output
flush	Flushes a stream	Output
hex	Turns on hexflag	Input/Output
internal	Turns on internalflag	Output
left	Turns on leftflag	Output
noboolalpha	Turns off boolalphaflag	Input/Output

noshowbase	Turns off showbaseflag	Output
noshowpoint	Turns off showpointflag	Output
noshowpos	Turns off showposflag	Output
noskipws	Turns off skipwsflag	Input
nounitbuf	Turns off unitbufflag	Output
nouppercase	Turns off uppercaseflag	Output
oct	Turns on octflag	Input/Output
resetiosflags(fmtflads f)	Turns off the flags specified in f	Input/Output
right	Turns on rightflag	Output
scientific	Turns on scientificflag	Output
setbase(int base)	Sets the number base to base	Input/Output
setfill(int ch)	Sets the fill char ch	Output
setiosflags(fmtflags f)	Turns on the flags specified by f	Input/Output
setprecision(int p)	Sets the number of digits of precision	Output
setw(int w)	Sets the field width to w	Output
showbase	Turns on showbaseflag	Output
showpoint	Turns on showpointflag	Output
showpos	Turns on showposflag	Output
skipws	Turns on skipwsflag	Input
unitbuf	Turns on unitbuf	Output
uppercase	Turns on uppercaseflag	Output
ws	Skips leading white space	Input

The following program demonstrates several of the I/O manipulators:

```
#include<iostream>

#include<iomanip>

using namespacestd;

int main( ) {

cout<< hex << 100 << endl;

cout<< oct<< 10 << endl;

cout<< setfill('X') << setw(10);

cout<< 100 << " hi " << endl;

return0;
```

}

This program displays the following:

64

13

XXXXXXX144 hi

File I/O

A file is a bunch of bytes stored on some storage devices like hard disk, floppy disk etc. File I/O and console I/O are closely related. In fact, the same class hierarchy that supports console I/O also supports the file I/O. To perform file I/O, you must include <fstream> in your program. It defines several classes, including ifstream, ofstream and fstream. In C++, a file is opened by linking it to a stream. There are three types of streams: input, output and input/output. Before you can open a file, you must first obtain a stream.

To create an input stream, declare an object of type ifstream.

To create an output stream, declare an object of type ofstream.

To create an input/output stream, declare an object of type fstream.

For example, this fragment creates one input stream, one output stream and one stream capable of both input and output:

ifstream in; // input;

fstream out; // output;

fstream io;   // input and output

Once you have created a stream, one way to associate it with a file is by using the function open( ). This function is a member function of each of the three stream classes. The prototype for each is shown here:

void ifstream::open(const char*filename,openmode mode=ios::in);

void ofstream::open(const char*filename,openmode mode=ios::out | ios::trunc);

void fstream::open(const char*filename,openmode mode=ios::in | ios::out);

Here filename is the name of the file, which can include a path specifier. The value of the mode determines how the file is opened. It must be a value of type open mode, which is an enumeration defined by ios that contains the following value:

- ios::app: causes all output to that file to be appended to the end. Only with files capable of output.
- ios::ate: causes a seek to the end of the file to occur when the file is opened.
- ios::out: specify that the file is capable of output.
- ios::in: specify that the file is capable of input.
- ios::binary: causes the file to be opened in binary mode. By default, all files are opened in text mode. In text mode, various character translations might take place, such as carriage return/linefeed

sequences being converted into newlines. However, when a file is opened in binary mode, no such character translations will occur.

• ios::trunc: causes the contents of a pre-existing file by the same name to be destroyed and the file to be truncated to zero length. When you create an output stream using ofstream, any pre-existing file is automatically truncated.

All these flags can be combined using the bitwise operator OR (|). For example, if we want to open the file example.bin in binary mode to add data we could do it by the following call to member function open:

ofstream myfile;

myfile.open ("example.bin", ios::out | ios::app | ios::binary);
Each of the open member functions of classes ofstream, ifstream and fstream has a default mode that is used if the file is opened without a second argument:

Class	default mode parameter
ofstream	ios::out
ifstream	ios::in
fstream	ios::in | ios::out

Closing file

When we are finished with our input and output operations on a file we shall close it so that the operating system is notified and its resources become available again. For that, we call the stream's member function `close`. This member function takes flushes the associated buffers and closes the file:

myfile.close();

Once this member function is called, the stream object can be re-used to open another file, and the file is available again to be opened by other processes. In case that an object is destroyed while still associated with an open file, the destructor automatically calls the member function close.

To write to a file, you construct a `ofsteam` object connecting to the output file, and use the `ostream` functions such as stream insertion `<<`, `put()` and `write()`. Similarly, to read from an input file, construct an `ifstream` object connecting to the input file, and use the `istream` functions such as stream extraction `>>`, `get()`, `getline()` and `read()`.

There are two ways of storing data in a file as given below:

Binary form and Text form

Suppose, we want to store a five digit number say 19876 in the text form, then it will be stored as five characters. Each character occupies one byte, which means that we will require five bytes to store five-digit integer in the text form. This requires storage of 40 bits. Therefore, if we can store them in binary form, then we will need only two bytes or 16 bits to store the number. The savings will be much more when we deal with floating point numbers.

When we store a number in text form, we convert the number to characters. However, storing a number in binary form requires storing it in bits. However, for a character, the binary representation as well as the text representation are one and the same since, in either case, it occupies eight bits. The text format is easy to read. We can even use a notepad to read and edit a text file. The portability of text file is also assured. In case of numbers, binary form is more appropriate. It also occupies lesser space when we store it in binary form and hence it will be faster. The default mode is text.

Unformatted, binary I/O

C++ supports a wide range of unformatted file I/O functions. The unformatted functions give you detailed control over how files are written and read. The lowest-level unformatted I/O functions are get( )and put( ). You can read a byte by using get( ) and write a byte by using put( ). These functions are member functions of all input and output stream classes, respectively. The get( )function has many forms, but the most commonly used version is shown here, along with put( ):

istream &get(char&ch);

ostream &put(char&ch);

To read and write blocks of data, use read( )and write()functions, which are also member functions of the input and output stream classes, respectively. Their prototypes are:

istream &read(char*buf, streamsize num);

ostream &write(const char*buf, streamsize num);

The read( ) function reads num bytes from the stream and puts them in the buffer pointed to by buf. The write() function writes num bytes to the associated stream from the buffer pointed by buf. The streamsize type is some form of integer. An object of type streamsize is capable of holding the largest number of bytes that will be transferred in any I/O operation. If the end of file is reached before num characters have been read, read( ) stops and the buffer contains as many characters as were available.

When you are using the unformatted file functions, most often you will open a file for binary rather than text operations. The reason for this is easy to understand: specifying ios::binary prevents any character translations from occurring. This is important when the binary representations of data such as integers, floats and pointers are stored in the file. However, it is perfectly acceptable to use the unformatted functions on a file opened in text mode, as long as that the file actually contains only text. But remember, some character translation may occur.

Random Access

File pointers

C++ also supports file pointers. A file pointer points to a data element such as character in the file. The pointers are helpful in lower level operations in files. There are two types of pointers:

get pointer

put pointer

The get pointer is also called input pointer. When we open a file for reading, we can use the get pointer. The put pointer is also called output pointer. When we open a file for writing, we can use put pointer. These pointers are helpful in navigation through a file. When we open a file for reading, the get pointer will be at location zero and not 1. The bytes in the file are numbered from zero. Therefore, automatically when we assign an object to ifstream and then initialize the object with a file name, the get pointer will be ready to read the contents from 0^{th} position. Similarly, when we want to write we will assign to an ofstream object a filename. Then, the put pointer will point to the 0^{th} position of the given file name after it is created. When we open a file for appending, the put pointer will point to the

0^{th} position. But, when we say write, then the pointer will advance to one position after the last character in the file.

File pointer functions

There are essentially four functions, which help us to navigate the file as given below

Functions Function Purpose

tellg()	Returns the current position of the get pointer
seekg()	Moves the get pointer to the specified location
tellp()	Returns the current position of the put pointer
seekp()	Moves the put pointer to the specified location

```
//To demonstrate writing and reading-
using open #include<fstream.>
#include<iostream>
int main(){

//Writing ofstream outf;
outf.open(“Temp2.txt”);
outf<<“Working with files
is fun\n”; outf<<“Writing
to files is also fun\n”;
outf.close();
char
buff[80];
ifstream
inf;
inf.open(“Temp2.txt”);                    //Reading while(inf){
inf.getline
(buff, 80);
cout<<buf
f<<“\n”;
}
inf.close(); return 0; }
```

Unit-V

STREAMS AND FILES

Unit Structure

Input Output With Files

Methods of Input and Output Classes

Text mode files

State flags

get and put stream pointers

Binary files

I/O Manipulators

INPUT OUTPUT WITH FILES

The techniques for file input and output, i/o, in C++ are virtually identical to those

introduced in earlier lessons for writing and reading to the standard output devices, the screen and keyboard. To perform file input and output the include file fstream must be used.

#include <fstream>

Fstream contains class definitions for classes used in file i/o.

Within a program needing file i/o, for each output file required, an object of class ofstream is instantiated.

For each input file required, an object of class ifstream is instantiated.

The ofstream object is used exactly as the cout object for standard output is used.

The ifstream object is used exactly as the cin object for standard input is used.

This is best understood by studying an example.

C++ has support both for input and output with files through the following classes:

ofstream: File class for writing operations (derived from ostream)

ifstream : File class for reading operations (derived from istream)

fstream : File class for both reading and writing operations (derived

from iostream)

Open a file

The first operation generally done on an object of one of these classes is to associate it to a real file, that is to say, to open a file. The open file is represented within the program by a stream object (an instantiation of one of these classes) and any input or output performed on this stream object will be applied to the physical file.

In order to open a file with a stream object we use its member function open():

void open (const char * filename, openmode mode);

where filename is a string of characters representing the name of the file to be opened and mode is a combination of the following flags:

ios::in Open file for reading ios::out Open file for writing ios::ate Initial position: end of file

ios::app Every output is appended at the end of file ios::trunc If the file already existed it is erased ios::binary Binary mode

These flags can be combined using bitwise operator OR:

|. For example, if we want to open the file "example.bin" in binary mode to add data we could do it by the following call to function- member

open:

ofstream file;

file.open ("example.bin", ios::out | ios::app | ios::binary);

All of the member functions open of classes ofstream, ifstream and fstream include a default mode when opening files that varies from one to the other:

class	default mode to parameter	
ofstream	ios::out | ios::trunc ifstream	ios::in
fstream	ios::in | ios::out	

The default value is only applied if the function is called without specifying a mode parameter. If the function is called with any value in that parameter the default mode is stepped on, not combined.

Since the first task that is performed on an object of classes **ofstream, ifstream** and **fstream** is frequently to open a file, these three classes include a constructor that directly calls the **open** member function and has the same parameters as this. This way, we could also have declared the previous object and conducted the same opening operation just by writing: ofstream file ("example.bin", ios::out | ios::app | ios::binary); Both forms to open a file are valid.

You can check if a file has been correctly opened by calling the member function **is_open()**:

bool is_open(); that returns a **bool** type value indicating **true** in case that indeed the object has been correctly associated with an open file or **false** otherwise.

Closing a file

When reading, writing or consulting operations on a file are complete we must close it so that it becomes available again. In order to do that we shall call the member function **close()**, that is in charge of flushing the buffers and closing the file. Its form is quite simple:

void close ();

Once this member function is called, the stream object can be used to open another file, and the file is available again to be opened by other processes.

In case that an object is destructed while still associated with an open file, the destructor automatically calls the member function **close**.

METHODS OF INPUT AND OUTPUT CLASSES

The ifstream class has several useful methods for input. These method are also in the class **cin**, which is used to read from standard input. These methods are used to read from any input stream. An input stream is a source of input such as the keyboard, a file or a buffer.

1) Extraction Operator >>

This overloaded operator handles all built in C++ data types. By default, any intervening white space is disregarded. That is, blanks, tabs, new lines, formfeeds and carriage returns are skipped over.

2) get()

This form of get extracts a single character from the input stream, that is, from the standard input, a file or a buffer. It does not skip white space. It returns type int.

3) get(char &ch)

This form of get also extracts a single character from the input stream, but it stores the value in the character variable passed in as an argument.

4) get(char *buff, int buffsize, char delimiter='\n')

This form of get reads characters into the C-style buffer passed as an argument buffsize characters are read, the delimiter is encountered or an end of file is encountered. The '\n' is the new line character. The delimiter is not read into the buffer but is instead left in the input stream. It must be removed separately but using either another get or an ignore. Because of this added step, this form of get is a frequent source of errors and should be avoided. Fortunately, another method shown below, getline, reads in the delimiter as well and should be used in place of this form of get.

5) Getline

There are several useful forms of getline.

ignore(int count=1, int delim=traits_type::eof)

This method reads and discards "count" characters from the input stream.

6) peek()

This method returns the next character from the input stream, but does not remove it. It is useful to look ahead at what the next character read will be.

7) putback(char &ch)

This method puts ch onto the input stream. The character in ch will then be the next character read from the input stream.

8) unget()

This method puts the last read character back into the input stream.

9) read(char *buff, int count)

This method is used to perform an unformatted read of count bytes from the input stream into a character buffer.

The ofstream class has several useful methods for writing to an output stream. An output stream is standard output (usually the screen), a file or a buffer. These methods are also in the object cout, which is used for standard output. The simplest way to understand how to use these methods is by looking at a few examples. Since we have seen the extraction, >>, and insertion, << in several lessons, let's look at the other methods. Getline, which is very useful to read entire lines of text into a string.

Suppose we need to read a file and determine the number of alphanumeric characters, the number of blanks and the number of sentences. To determine the number of sentences we will count the number of periods (dots). We will disregard newlines and tabs.

Here is a program that solves the problem.

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
int blank_count = 0; int char_count = 0;
int sentence_count = 0; char ch;
ifstream iFile("c:/lesson12.txt"); if (! iFile)
{
cout << "Error opening input file" << endl; return -1;
}
while (iFile.get(ch))
{
switch (ch)
{
case ' ': blank_count++; break;

case '\n':
case '\t': break;
```

```
case '.': sentence_count++; break;

default: char_count++; break;
}
endl;}
cout <<  "There  are " << blank_count <<  " blanks"   <<

cout << "There are " << char_count << " characters"
<< endl;
cout << "There are " << sentence_count << " sentences" << endl;
return 0;
}
```

As a second example, let's implement a program that will copy the contents of one file to another. The program will prompt the user for the input and output file names, and then copy.

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

int main()
{
char ch;
string iFileName; string oFileName;
cout << "Enter the source file name: "; cin >> iFileName;
cout << "Enter the destination file name: "; cin >> oFileName;
ofstream oFile(oFileName.c_str()); ifstream iFile(iFileName.c_str());
//Error checking on file opens omitted for brevity. while (iFile.get(ch))
{
oFile.put(ch);
}
return 0;
}
```

TEXT MODE FILES

Classes ofstream, ifstream and fstream are derived from ostream, istream and iostream respectively. That's why fstream objects can use the members of these parent classes to access data.

Generally, when using text files we shall use the same members of these classes that we used in communication with the console (cin and cout).

As in the following example, where we use the overloaded insertion operator <<:

```
// writing on a text file #include <fstream.h> int main ()
{
ofstream examplefile ("example.txt"); if (examplefile.is_open())
{
examplefile << "This is a line.\n"; examplefile << "This is another line.\n"; examplefile.close();
}
return 0;
}
```

The output is : This is a line.

This is another line.

Data input from file can also be performed in the same way that we did with cin:

```
// reading a text file #include <iostream.h> #include <fstream.h> #include <stdlib.h>
int main ()
{
char buffer[256];
ifstream examplefile ("example.txt"); if (! examplefile.is_open())
{ cout << "Error opening file"; exit (1); } while (! examplefile.eof() )
{
examplefile.getline (buffer,100); cout << buffer << endl;
}
return 0;
}
```

The output is : This is a line. This is another line.

This last example reads a text file and prints out its content on the screen. Notice how

we have used a new member function, called eof that ifstream inherits from class ios and that returns true in case that the end of the file has been reached.

STATE FLAGS

In addition to eof(), other member functions exist to verify the state of the stream (all of them return a bool value):

1) bad() : Returns true if a failure occurs in a reading or writing operation. For example in case we try to write to a file that is not open for writing or if the device where we try to write has no space left.

2) fail() : Returns true in the same cases as bad() plus in case that a format error happens, as trying to read an integer number and an alphabetical character is received.

3) eof() : Returns true if a file opened for reading has reached the end.

4) good() : It is the most generic: returns false in the same cases in which calling any of the previous functions would return true.

In order to reset the state flags checked by the previous member functions you can use member function clear(), with no parameters.

GET AND PUT STREAM POINTERS

All i/o streams objects have, at least, one stream pointer:

1) **ifstream**, like **istream**, has a pointer known as get pointer that points to the next element to be read.

2) **ofstream**, like **ostream**, has a pointer put pointer that points to the location where the next element has to be written.

3) **fstream**, like **iostream**, inherits both: get and put

These stream pointers that point to the reading or writing locations within a stream can be read and/or manipulated using the following member functions:

tellg() and tellp()

These two member functions admit no parameters and return a value of type pos_type (according ANSI-C++ standard) that is an integer data type representing the current position of get stream pointer (in case of tellg) or put stream pointer (in case of tellp).

seekg() and seekp()

This pair of functions serve respectively to change the position of stream pointers get and put. Both functions are overloaded with two different prototypes:

seekg (pos_type position); seekp (pos_type position);

Using this prototype the stream pointer is changed to an absolute position from the beginning of the file. The type required is the same as that returned by functions tellg and tellp.

seekg (off_type offset, seekdir direction); seekp (off_type offset, seekdir direction);

Using this prototype, an offset from a concrete point determined by parameter direction can be specified. It can be:

ios::beg Offset specified from the beginning of the stream ios::cur Offset specified from the current position of the stream pointer

ios::end Offset specified from the end of the stream

The values of both stream pointers get and put are counted in different ways for text files than for binary files, since in text mode files some modifications to the appearance of some special characters can occur. For that reason it is advisable to use only the first prototype of **seekg** and

seekp with files opened in text mode and always use nonmodified values returned by **tellg** or **tellp**. With binary files, you can freely use all the implementations for these functions. They should not have any unexpected behavior.

The following example uses the member functions just seen to obtain the size of a binary file:

```
// obtaining file size #include <iostream.h> #include <fstream.h>

const char * filename = "example.txt";

int main ()

{

long l,m;

ifstream file (filename, ios::in|ios::binary); l = file.tellg();


file.seekg (0, ios::end); m = file.tellg(); file.close();

cout << "size of " << filename;

cout << " is " << (m-l) << " bytes.\n"; return 0;

}
```

The output is : size of example.txt is 40 bytes.

BINARY FILES

In binary files inputting and outputting data with operators like << and >> and functions like **getline**, does not make too much sense, although they are perfectly valid. File streams include two member functions specially designed for input and output of data sequentially: **write** and **read**. The first one (**write**) is a member function of **ostream**, also inherited by **ofstream**. And **read** is member function of **istream** and it is inherited by **ifstream**. Objects of class **fstream** have both. Their prototypes are:

write (char * buffer, streamsize size); read (char * buffer, streamsize size);

Where buffer is the address of a memory block where the read data are stored or from

where the data to be written are taken. The size parameter is an integer value that specifies the number of characters to be read/written from/to the buffer.

```
// reading binary file #include <iostream.h> #include <fstream.h>

const char * filename = "example.txt";

int main ()

{

char * buffer; long size;

ifstream file (filename, ios::in|ios::binary|ios::ate); size = file.tellg(); file.seekg (0, ios::beg); buffer = new char [size];


file.read (buffer, size); file.close();
```

```
cout << "the complete file is in a buffer"; delete[] buffer;
return 0;
}
```

The output is : the complete file is in a buffer

BUFFERS AND SYNCHRONIZATION

When we operate with file streams, these are associated to a buffer of type streambuf.

This buffer is a memory block that acts as an intermediary between the stream and the physical file. For example, with an out stream, each time the member function put (write a single character) is called, the character is not written directly to the physical file with which the stream is associated. Instead of that, the character is inserted in thebuffer for that stream.

When the buffer is flushed, all data that it contains is written to the physic media (if it is an out stream) or simply erased (if it is an in stream). This process is called synchronization and it takes place under any of the following circumstances:

When the file is closed: before closing a file all buffers that have not yet been completely written or read are synchronized.

When the buffer is full: Buffers have a certain size. When the buffer is full it is automatically synchronized.

Explicitly with manipulators: When certain manipulators are used on streams synchronization takes place. These manipulators are: flush and endl.

Explicitly with function sync(): Calling member function sync() (no parameters) causes an immediate syncronization. This function returns an int value equal to -1 if the stream has no associated buffer or in case of failure.

I/O MANIPULATORS

Up till now, we have accepted the default output formatting. C++ defines a set of manipulators which are used to modify the state of iostream objects. These control how data is formatted. They are defined in the include file, <ios>. It is not usually necessary to explicitly include this file because it is included indirectly via the use of other includes such as <iostream> or

<fstream>.

Let's see how some of these manipulators work in a simple program.

Manipulator	**Use**
Boolalpha	Causes bool variables to be output as true or false.
Noboolalhpa (default)	Causes bool variables to be displayed as 0 or 1
Dec (default)	Specifies that integers are displayed in base 10.
Hex	Specifies that integers are displayed in hexadecimal.
Oct	Specified that integers are displayed in octal.
Left	Causes text to be left justified in the output field.
Right	Causes text to be right justified in the output field.
Internal	Causes the sign of a number to be left justified and the value to be right justified.
Noshowbase (default)	Turns off displaying a prefix indicating the base of a number.
Showbase	Turns on displaying a prefix indicating the base of a number.
Noshowpoint (default)	Displays decimal point only if a fractional part exists.
Showpoint	Displays decimal point always.
noshowpos (default)	No "+" prefixing a positive number.
Showpos	Displays a "+" prefixing a positive number.
Skipws (default)	Causes white space (blanks, tabs, newlines) to be skipped by the input operator, >>.
Noskipws	White space not skipped by the extraction operator, >>.
Fixed (default)	Causes floating point numbers to be displayed in fixed notation.
Scientific	Causes floating point numbers to be displayed in scientific notation.

Nouppercase (default)	0x displayed for hexadecimal numbers, e for scientific notation
Uppercase	0X displayed for hexadecimal numbers, E for scientific notation

The manipulators in the above table modify the state of the iostream object. This means that once used on an iostream object they will effect all subsequent input or output done with the object. There are several other manipulators that are used to format a particular output but do no modify the state of the object.

Setting Output Width

setw(w) - sets output or input width to w; requires <iomanip> to be included.

width(w) - a member function of the iostream classes.

Filling White Space

setfill(ch) - fills white space in output fields with ch; requires

<iomanip> to be included.

fill(ch) = a member function of the iostream classes.

Setting Precision

setprecision(n) - sets the display of floating point numbers at precision n. This does not

effect the way floating point numbers are handled during calculations in your program.

Here is a simple program illustrating the use of the i/o manipulators. #include <iostream>

#include <iomanip> #include <string> using namespace std; int main()

{

int intValue = 15;

cout << "Integer Number" << endl; cout << "Default: " << intValue << endl;

cout << "Octal: " << oct << intValue << endl;

endl; endl;

```
cout << "Hex: " << hex << intValue << endl;

cout << "Turning showbase on" << showbase <<

cout << "Dec: " << dec << intValue << endl; cout << "Octal: " << oct <<

intValue << endl; cout << "Hex: " << hex << intValue << endl;

cout << "Turning showbase off" << noshowbase <<

cout << endl;

double doubleVal = 12.345678;

cout << "Floating Point Number" << endl; cout << "Default: " << doubleVal << endl; cout << setprecision(10);

cout << "Precision of 10: " << doubleVal << endl;

cout << scientific << "Scientific Notation: " <<

doubleVal << endl;

cout << uppercase;

cout << "Uppercase: " << doubleVal << endl; cout << endl;

bool theBool = true;

cout << "Boolean" << endl;

cout << "Default: " << theBool << endl;

cout << boolalpha << "BoolAlpha set: " << theBool <<

endl;

        cout << endl;

        string myName = "John"; cout << "Strings" << endl;

        cout << "Default: " << myName << endl;

        cout << setw(35) << right << "With setw(35) and right:

"<< myName << endl;

cout.width(20);

cout << "With width(20): " << myName << endl; cout << endl;

return 0;

}
```

TEMPLATES

Unit Structure :

Function templates

Class templates

FUNCTION TEMPLATES

Function templates are special functions that can operate with *generic types*. This allows us to create a function template whose functionality can be adapted to more than one type or class without repeating the entire code for each type.

In C++ this can be achieved using *template parameters*. A template parameter is a special kind of parameter that can be used to pass a type as argument: just like regular function parameters can be used to pass values to a function, template parameters allow to pass also types to a function. These function templates can use these parameters as if they were any other regular type.

The format for declaring function templates with type parameters is: template <class identifier> function_declaration; template <typename identifier> function_declaration;

The only difference between both prototypes is the use of either the keyword class or the keyword typename. Its use is indistinct, since both expressions have exactly the same meaning and behave exactly the same way.

For example, to create a template function that returns the greater one of two objects we could use:

```
template <class myType>
myType GetMax (myType a, myType b)
{
return (a>b?a:b);
}
```

Here we have created a template function with myType as its template parameter. This template parameter represents a type that has not yet been specified, but that can be used in the template function as if it were a regular type. As you can see, the function template GetMax returns the greater of two parameters of this still-undefined type.

To use this function template we use the following format for the function call:

```
function_name <type> (parameters);
```

For example, to call GetMax to compare two integer values of type int we can write:

```
int x,y;
GetMax <int> (x,y);
```

When the compiler encounters this call to a template function, it uses the template to automatically generate a function replacing each appearance of myType by the type passed as the actual template parameter (int in this case) and then calls it. This process is automatically performed by the compiler and is invisible to the programmer.

Here is the entire example:

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

template <class T> T GetMax (T a, T b)
{
T result;
result = (a>b)? a : b; return (result);
}

int main ()
{
int i=5, j=6, k; long l=10, m=5, n;
k=GetMax<int>(i,j); n=GetMax<long>(l,m); cout << k << endl; cout << n << endl; return 0;
}
```

The output is:

10

In this case, we have used T as the template parameter name instead of myType because it is shorter and in fact is a very common template parameter name. But you can use any identifier you like.

In the example above we used the function template GetMax() twice. The first time with arguments of type int and the second one with arguments of type long. The compiler has instantiated and then called each time the appropriate version of the function.

As you can see, the type T is used within the GetMax() template function even to declare new objects of that type:

T result;

Therefore, result will be an object of the same type as the parameters a and b when the function template is instantiated with a specific type.

In this specific case where the generic type T is used as a parameter for GetMax the compiler can find out automatically which data type has to instantiate without having to explicitly specify it within angle brackets (like we have done before specifying <int> and <long>). So we could have written instead:

int i,j; GetMax (i,j);

Since both i and j are of type int, and the compiler can automatically find out that the template parameter can only be int. This implicit method produces exactly the same result:

```
// function template II #include <iostream> using namespace std;

template <class T> T GetMax (T a, T b)
{
return (a>b?a:b);
}
int main ()
{
int i=5, j=6,  k; long l=10, m=5, n; k=GetMax(i,j); n=GetMax(l,m); cout << k << endl;
                          cout << n << endl; return 0;

}
```

6

10

The output is :

Notice how in this case, we called our function template GetMax() without explicitly specifying the type between angle- brackets <>. The compiler automatically determines what type is needed on each call.

Because our template function includes only one template parameter (class T) and the function template itself accepts two parameters, both of this T type, we cannot call our function template with two objects of different types as arguments:

```
int i; long l;
k = GetMax (i,l);
```

This would not be correct, since our GetMax function template expects two arguments of the same type, and in this call to it we use objects of two different types.

We can also define function templates that accept more than one type parameter, simply by specifying more template parameters between the angle brackets. For example:

```
template <class T, class U> T GetMin (T a, U b)
{
return (a<b?a:b);
}
```

In this case, our function template GetMin() accepts two parameters of different types and returns an object of the same type as the first parameter (T) that is passed. For example, after that declaration we could call GetMin() with:

```
int i,j; long l;
i = GetMin<int,long> (j,l); or simply:
i = GetMin (j,l);
```

even though j and l have different types, since the compiler can determine the appropriate instantiation anyway.

CLASS TEMPLATES

We also have the possibility to write class templates, so that a class can have members that use template parameters as types. For example:

```
template <class T> class mypair
{
```

```
T values [2]; public:
mypair (T first, T second)
{
values[0]=first; values[1]=second;
}
};
```

The class that we have just defined serves to store two elements of any valid type. For example, if we wanted to declare an object of this class to store two integer values of type int with the values 115 and 36 we would write:

```
mypair<int> myobject (115, 36);
```

this same class would also be used to create an object to store any other type:

```
mypair<double> myfloats (3.0, 2.18);
```

The only member function in the previous class template has been defined inline within the class declaration itself. In case that we define a function member outside the declaration of the class template, we must always precede that definition with the template

<...> prefix:

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

template <class T> class mypair
{
T a, b; public:
mypair (T first, T second)
{

a=first; b=second;
}
T getmax ();
};

template <class T>
T mypair<T>::getmax ()
{
```

```
T retval;

retval = a>b? a : b; return retval;

}


int main ()

{

mypair <int> myobject (100, 75); cout << myobject.getmax(); return 0;

}
```

The output is:

100

Notice the syntax of the definition of member function getmax:

```
template <class T>

T mypair<T>::getmax ()
```

Confused by so many T's? There are three T's in this declaration: The first one is the template parameter. The second T refers to the type returned by the function. And the third T (the one between angle brackets) is also a requirement: It specifies that this function's template parameter is also the class template parameter.

EXCEPTIONS

Unit Structure:

Exceptions

Exception specifications

EXCEPTIONS

Exceptions provide a way to react to exceptional circumstances (like runtime errors) in our program by transferring control to special functions called *handlers*.

To catch exceptions we must place a portion of code under exception inspection. This is done by enclosing that portion of code in a *try block*. When an exceptional circumstance arises within that block, an exception is thrown that transfers the control to the exception handler. If no exception is thrown, the code continues normally and all handlers are ignored.

A exception is thrown by using the throw keyword from inside the try block. Exception handlers are declared with the keyword catch, which must be placed immediately after the try block:

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

int main ()
{
try
{
                    throw 20;
}
catch (int e)
{
cout << "An exception occurred. Exception Nr.
" << e << endl;
}
return 0;
}
```

The output is :

An exception occurred. Exception Nr. 20

The code under exception handling is enclosed in a try block. In this example this code simply throws an exception:

```
throw 20;
```

A throw expression accepts one parameter (in this case the integer value 20), which is passed as an argument to the exception handler.

The exception handler is declared with the catch keyword. As you can see, it follows immediately the closing brace of the try block. The catch format is similar to a regular function that always has at least one parameter. The type of this parameter is very important, since the type of the argument passed by the throw expression is checked against it, and only in the case they match, the exception is caught.

We can chain multiple handlers (catch expressions), each one with a different parameter type. Only the handler that matches its type with the argument specified in the throw statement is executed.

If we use an ellipsis (...) as the parameter of catch, that handler will catch any exception no matter what the type of the throw exception is. This can be used as a default handler that catches all exceptions not caught by other handlers if it is specified at last:

```
try
{
        // code here
}
```

```
catch (int param)
{
cout << "int exception";
}
catch (char param)
{
cout << "char exception";
}
catch (...)
{
cout << "default exception";
}
```

In this case the last handler would catch any exception thrown with any parameter that is neither an int nor a char.

After an exception has been handled the program execution resumes after the try-catch block, not after the throw statement!.

It is also possible to nest try-catch blocks within more external try blocks. In these cases, we have the possibility that an internal catch block forwards the exception to its external level. This is done with the expression throw; with no arguments. For example:

```
try
{
try
{
                                    // code here
}
```

```
catch (int n)
{
throw;
}
}

catch (...)
{
cout << "Exception occurred";
}
```

EXCEPTION SPECIFICATIONS

When declaring a function we can limit the exception type it might directly or indirectly throw by appending a throw suffix to the function declaration:

```
float myfunction (char param) throw (int);
```

This declares a function called myfunction which takes one agument of type char and returns an element of type float. The only exception that this function might throw is an exception of type int. If it throws an exception with a different type, either directly or indirectly, it cannot be caught by a regular int-type handler.

If this throw specifier is left empty with no type, this means the function is not allowed to throw exceptions. Functions with no throw specifier (regular functions) are allowed to throw exceptions with any type:

int myfunction (int param) throw(); // no exceptions allowed int myfunction (int param); // all exceptions allowed

STANDARD EXCEPTIONS

The C++ Standard library provides a base class specifically designed to declare objects to be thrown as exceptions. It is called exception and is defined in the <exception> header file under the namespace std. This class has the usual default and copy constructors, operators and destructors, plus an additional virtual member function called what that returns a null-terminated character sequence (char *) and that can be overwritten in derived classes to contain some sort of description of the exception.

Example

```
// standard exceptions #include <iostream> #include <exception> using namespace std;

class myexception: public exception
{
virtual const char* what() const throw()
{
                                    return "My exception happened";
}
} myex;

int main ()
{
try
{
                                    throw myex;
}
```

```
catch (exception& e)
{
cout << e.what() << endl;
}
return 0;
}
```

The output is :

My exception happened.

We have placed a handler that catches exception objects by reference (notice the ampersand & after the type), therefore this catches also classes derived from exception, like our myex object of class myexception.

All exceptions thrown by components of the C++ Standard library throw exceptions derived from this std::exception class. These are:

Exception	Description
bad_alloc	thrown by new on allocation failure
bad_cast	thrown by dynamic_cast when fails with a referenced type
bad_exception	thrown when an exception type doesn't match any catch
bad_typeid	thrown by typeid
ios_base::failure	thrown by functions in the iostream library

For example, if we use the operator new and the memory cannot be allocated, an exception of type bad_alloc is thrown:

```
try
{
int * myarray= new int[1000];
}
catch (bad_alloc&)
{
cout << "Error allocating memory." << endl;
```

```
}
```

It is recommended to include all dynamic memory allocations within a try block that catches this type of exception to perform a clean action instead of an abnormal program termination, which is what happens when this type of exception is thrown and not caught. If you want to force a bad_alloc exception to see it in action, you can try to allocate a huge array; On my system, trying to allocate 1 billion ints threw a bad_alloc exception.

Because bad_alloc is derived from the standard base class exception, we can handle that same exception by catching references to the exception class:

```
// bad_alloc standard exception #include <iostream>

#include <exception> using namespace std;

int main ()
{
try
{
                int* myarray= new int[1000];
}
catch (exception& e)
{
        cout << "Standard exception: " << e.what() <<
endl;
        }
return 0;
}
```

THE C++ STANDARD TEMPLATE LIBRARY

Unit Structure :

INTRODUCTION

This is a short primer on the **STL** or "Standard Template Library" for the programming language C++ as defined in the 1997 standard. The **STL** is a collection C++ libraries that allow you to use several well known kinds of data structures with out having to program them. They are designed so that the code runs efficiently. The compiler does most of the work of generating the efficient implementations. The libraries include a large number of possibilities. we describe some of things you can do with the STL versions of stacks, queues, vectors, and lists. I define some important and useful ideas in the **Glossary** below. The is a table summarizing the methods used with stacks, queues, vectors and lists at **Summary** below.For each kind of data structure I give working examples of how to declare, access and use objects of that type.

STANDARD TEMPLATES

The international and American Standard for C++ requires there to be special libraries that implement the following generic data structures:

Library	**Description**
<**vector**>	A dynamic array [**$Vectors**]
<**list**>	A randomly changing sequence of items
<**stack**>	A sequence of items with **pop** and **push** at one end only
<**queue**>	A Sequence of items with **pop** and **push** at opposite ends
<deque>	Double Ended Queue with pop and push at both ends
<bitset>	A subset of of a fixed and small set of items
<set>	An unordered collection of items
<map>	An collection of pairs of items indexed by the first one

These are designed to be general, efficient, and powerful rather than easy to use.

. . . . (end of section **Standard Templates**) <<Contents | End>>

STACKS

Stacks are only accessed at their top. To be able to use **STL**
stacks in a file of C++ source code or a header file add #include <stack>
at the beginning of the file.

Suppose that *T* is any type or class - say an int, a float, a struct, or a class, then

stack<T> s;

declares a new and empty stack called *s*. Given *s* you can:

1) test to see if it is empty:

s.empty()

2) find how many items are in it:

s.size()

3) push a *t* of type *T* onto the top:

s.push(t)

4) pop the top off *s*: s.pop()

5) get the top item of *s*

s.top()

6) change the top item:

s.top() = expression.

Example of an STL Stack

```
void reverse(string & x)
//Afterwards x will have the same elements in the opposite
order.
{
        stack<char> s;
        const int n = x.length();
        //Put characters from x onto the stack for(int i=0; i<n; ++i)
        s.push(x[i]);
        //take characters off of stack and put them back into x for(int i=0; !s.empty();
        ++i, s.pop())
        x[i]=s.top();

}
```

OLDER STACKS

On some older C++ libraries you are forced to indicate how the stack is implemented whenever you declare one. You write either

```
stack< vector<T> > s;
```

or

```
stack< list<T> > s;
```

Example :

```
        void reverse(string & x) //INOUT: x is a string of
characters
        //Afterwards x will have the same elements in the
opposite order.
{
```

```
stack< vector<char> > s; const int n = x.length(); for(int i=0; i<n; ++i) s.push(x[i]);
for(int i=0; !s.empty(); ++i, s.pop()) x[i]=s.top();
}
```

. . . . (end of section **Stacks**) **<<**Contents | End**>>**

QUEUES

Queues allow data to be added at one end and taken out of the other end. To be able to use **STL** queues add this before you start using them in your source code:

```
#include <queue>
```

Suppose that *T* is any type or class - say an int, a float, a struct, or a class, then

1) declares a new and empty queue called *q*. Given an object *q*:

```
queue<T> q;
```

2) test to see if *q* is empty: q.empty()

3) find how many items are in *q*: q.size()

4) push a *t*:*T* onto the end of *q*: q.push(t)

5) pop the front of *q* off *q*: q.pop()

6) get the front item of *q*:

q.front()

7) change the front item: q.front() = expression.

8) get the back item of *q*:

q.back()

9) change the back item:

q.back() = expression.

Example of putting three items into a queue and then taking them off the queue.

```
#include <iostream.h> #include <list> #include <queue>
int main(){
```

```
queue<char> q; q.push('a');
q.push('b');
q.push('c');
cout << q.front(); q.pop();
cout << q.front(); q.pop();
cout << q.front(); q.pop();
}
```

OLDER QUEUES

On some older C++ libraries you are forced to indicate how the queue is implemented whenever you declare one. You write

```
queue< list<T> > q;
```

in place of

```
queue< T > q;
```

Example on putting three items into a queue and then taking them off the queue.

```
#include <iostream.h> #include <list> #include <queue>
int main()
{
queue< list <char> >q; q.push('a');
q.push('b');
q.push('c');
cout << q.front(); q.pop();
cout << q.front(); q.pop();
cout << q.front(); q.pop();
}
```

. . . . (end of section **Queues**) **<<**Contents | End**>>**

LISTS

Lists are good when data needs to be reorganized a lot. To be able to use **STL** lists add this before you start using them in your source code:

#include <list>

Suppose that *T* is any type or class - say an int, a float, a struct, or a class, then

1) To declares a new and empty list called *l*. Given an object *l*: list<T> l;

2) test to see if *l* is empty:

l.empty()

3) find how many items are in *l*:

l.size()

4) push a *t*:*T* onto the end of *l*:

l.push_back(t)

5) pop the last off *l*:

l.pop_back()

6) push a *t*:*T* onto the start of *l*:

l.push_front(t)

7) pop the front of *l* off *l*:

l.pop_front()

8) get the front item of *l*:

l.front()

9) change the front item:

l.front() = expression.

10) get the back item of *l*:

l.back()

11) change the back item:

l.back() = expression.

12) Sort the list:

l.sort()

13) Clear the list:

l.clear()

14) Merge in a sorted list into a sorted list:

l.merge(list_of_sorted_elements)

15) Reverse the list:

l.reverse()

16) Assign a copy of *q1* to *q*:

q = q1 [**Lists and Iterators**]

Example of List Processing

This uses a utility function 'print' that is implemented below [**An Example of Iterating Through a List**]

Building and Sorting a List

```
//Using a list to sort a sequence of 9 numbers. #include <iostream.h>

#include <list>

// #include <algorithm>


void print(list<int> a);//utility function print lists of ints int main()

{

list<int> a;

//Put 9,8,7,6,5,4,3,2,1 onto the list for(int i=0; i<9;++i)

a.push_back(9-i);// put new element after all the others print(a); //here the list contains (9,8,7,6,5,4,3,2,1) a.sort();//in the <list> library!

print(a); //here the list contains (1,2,3,4,5,6,7,8,9)

}
```

. . . . (end of section **Lists**) **<<**Contents | End**>>**

VECTORS

Vectors are good when we have an unknown sequence of items to store but we want to access the by their sequence numbers. To be able to use **STL** vectors add this before you start using them in your source code:

#include <vector>

Suppose that *T* is any type or class **-** say an int, a float, a struct, or a class, then

1) to declares a new and empty **vector** called *v*. Given object *v*

declare like the above:

vector<T> v;

2) test to see if *v* is empty:

v.empty()

3) find how many items are in *v*:

v.size()

4) push a *t*:*T* onto the end of *v*:

v.push_back(t)

5) pop the front of *v* off *v*:

v.pop_back()

6) get the front item of *v*:

v.front()

7) change the front item:

v.front() = expression.

8) get the back item of v:

v.back()

9) change the back item:

v.back() = expression.

10) Access the *i*'th item (0<=*i*<size()) without checking to see if it exists:

v[i]

11) Access the *i*'th item safely:

v.at(i)

12) Assign a copy of *v1* to *v*:

v = v1

Sorting and operating on all items efficiently, see **Iterators** below.

. . . . (end of section **Vectors**) <<Contents | End>>

ITERATORS AND CONTAINERS

Containers:

A Container is a data structure that holds a number of object of the same type or class. The oldest example of a Container in C++ is an array. Even in C you had arrays and you would typically write code like this:

```
const int MAX = 10; float a[MAX];
...
for(int i=0; i<10; ++i)
process(a[i]);
...
```

Lists, **Vectors**, **Stacks**, **Queues**, etc are all **Containers**.

C arrays were designed so that they could be accessed by using a variable that stores the address of an item. These variables are called *pointer*s. They are used like this with an array:

```
for(float *p=a; p!=p+MAX; ++p) process(*p);
...
```

Iterators:

Items in **Containers** are referred to be special objects called: *iterator*s. They are generalization of C's pointers. With an iterator class *Iter* you can process each item in a vector or a list by similar code:

```
for( Iter p=c.begin(); p!=c.end(); ++p) process(*p);
```

All **Containers** C in the STL have a number of iterator objects. Container class C has an iterator called

C::iterator

For any type *T*, list<*T*> and vector<*T*> are **Containers**. So there are iterator classes called list<T>::iterator and they are used like this:

```
for( list<T>::iterator p=c.begin(); p!=c.end(); ++p) process(*p);
```

Vector iterators have type

vector<T>::iterator and used like this:

```
for( vector<T>::iterator p=c.begin(); p!=c.end(); ++p) process(*p);
```

If you change your choice from a **vector** to a **list** then the code is almost identical. This makes your code easier to modify.

For any **container** class *C* of objects type *T* and any object *c* of type *C*:

Declare an iterator for **container** of type C.

C::iterator p

Move iterator *p* onto the next object in *c*(if any!).

++p

The value selected by *p*.

*p

The iterator that refers to the first item in *c*

c.begin()

The iterator that refers to one beyond *c*.

c.end()

Declare an iterator for **container** of type *C* and set to the start of *c*.

C::iterator p = c.begin();

Test to see if iterator p has come to the end of object *c*: p != c.end();

assign *p1* to *p* -- afterwards both refer to same item p = p1;

Lists and Iterators

Insertion and deletion of items at the start or inside a List of elements is controlled by an iterator:

Insert item *x* into List *l* before iterator *p*

l.insert(p, x);

Erase the element pointed at by iterator *q* in List *l*

l.erase(q);

Example:

```
#include <iostream.h>


#include <list>
void print(list <char> );// elsewhere main()
{
list <char> l;
list <char>::iterator p; l.push_back('o');
l.push_back('a');
l.push_back('t'); p=l.begin();

cout <<" "<< *p<<endl; // p refers to the 'o' in
('o', 'a', 't')
print(l); l.insert(p, 'c');
// l is now ('c', 'o', 'a', 't') and p still refers to 'o' cout <<" "<< *p<<endl;
print(l);
l.erase(p);
cout <<" "<< *p<<endl; // p refers to an 'o' but it

print(l);
is not in l!
l.erase(l.begin()); //removes front of l print(l);


}
```

Ranges:

A pair of iterators describes a **range** of items in their **container**. The items in the **range** start with the first iterator in the pair. The **range** has all the following items up to just before the item referred to by the last iterator of the pair.

Suppose that *first* and *last* form a **range** and *it* is an iterator then: for(it=first; it != last; it++)

will refer to each element in the **range** [*first..last*) in turn. In the body of the for loop the value of the element is **it*.

Given a **container** *c*, then *c.begin()* and *c.end()* form a **range**.

An Example of Iterating Through a List

```
void print( list<int> a)
{
for(list<int>::iterator ai=a.begin(); ai!=a.end(); ++ai) cout << *ai << " ";
cout << endl;
cout << " ---------------------------------- "<<endl;
}
```

ALGORITHMS

Ranges and Iterators are used several useful functions and algorithms in the **STL**. Suppose you have a type or class of data called *T* and in a program are working with a vector *v* of type *vector<T>* then

vector<T>::iterator

is the class of suitable iterators. Here are two useful values: v.begin()

v.end()

These always form a **range** of all the items in *v* from the **front** up to and including the **back**. You can sort a **vector** v simply by writing:

```
sort(c.begin(), c.end());
```

Example : Sorting a vector with 9 integers

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

void print(vector<int>) ;//utility function outputs a $container int main()

{

vector<int> a;

// Place 9,8,7,6,5,4,3,2,1 into the vector for(int i=0; i<9;++i)

a.push_back(9-i);// put new element after all the others print(a); // elements of a are (9,8,7,6,5,4,3,2,1)

sort(a.begin(), a.end()); //in the STL <algorithm> library print(a); // elements are nor in order.

}

void print(vector<int> a)

{

for(vector<int>::iterator ai=a.begin(); ai!=a.end(); ++ai) cout << *ai << " ";

cout << endl;

cout << " ------------------------------------ "<<endl;

}

. . . . (end of section **Iterators**) <<Contents | End>> **Summary**

Templat e	**push/pop**	**Commo n**	**items**	**Extras**
stack	pop(),push(T)	empty(), size()	top()	-
queue	pop(),push(T)	empty(), size()	front(),back( )	-
vector	push_back(T), pop_back()	empty(), size()	front(), back()	[int], at(int), =
list	push_back(), pop_back(), push_front(T) , pop_front()	empty(), size()	front(), back()	sort(), clear(), reverse(), merge(l),at(int), =

Note:

You must always #include headers if you use the words: vector, list,string,queue, or stack in your program or in a header file.

Their must be a space between the two ">" signs below: stack< vector< T > > s;

If the standard <list> and <vector> is not found then you are using an older C++ compiler.

On some older compilers and currant libraries when you need a

<string> as well as either <list> or <vector> you need to #include <string>

before including the list or vector rather in the reverse order! Older string library appear to define some special versions of vector and list operators and the older compilers can not make up its mind which to use.