



KARPAGAM ACADEMY OF HIGHER EDUCATION
(Deemed University Established Under Section 3 of UGC Act 1956)
Coimbatore - 641021.

(For the candidates admitted from 2016 onwards)

DEPARTMENT OF COMMERCE (CA)

SYLLABUS

Semester III

		L	T	P	C
16CCU302	OBJECT ORIENTED PROGRAMMING WITH C++	4	-	-	4

Course Objective

- ❖ To understand the basic principles of Object Oriented Programming with C++ , Tokens , Expressions and Control Structures, Classes and Objects, Operator Overloading and Pointers and Virtual Functions .

Learning Outcomes

- To enriches the knowledge of students on the applicability of oops concepts
- To understand compilation process for the real time Applications.

UNIT I

Principles of Object- Oriented Programming – A Look at Procedure and Object - Oriented Paradigm – Basic Concepts of Object – Oriented Programming – Benefits of Oop – Object-Oriented Languages – Applications of Oop . Beginning with C++ - What is C++? – Applications of C++ - C++ Statements – Structure of C++ Program.

UNIT II

Tokens, Expressions and Control Structures – Tokens – Keywords – Identifiers – Basic & User – Defined Data Types – Operators in C++ - Operator Over Loading – Operator Precedence – Control Structures – Functions in C++ - The Main Function – Function

Prototyping – Call By Reference – Return By Reference – In Line Functions – Function Over Loading – Friend and Virtual Functions.

UNIT III

Classes and Objects – Introduction – Specifying a Class – Defining Member Function – Nesting of Member Functions – Private Member Functions – Arrays within a Class- Static Data Members – Static Member Functions – Array of Objects – Objects as Function Arguments – Friendly Functions – Pointers to Members. Constructors & Destructors – Constructors – Copy Constructors – Dynamic Constructors – Construction Two- Dimensional Arrays – Destructors.

UNIT IV

Operator Over Loading -Type Conversion – Introduction – Defining Operator Over Loading – Over Loading Unary & Binary Operators – Over Loading Binary Operators using Friends – Manipulation of String Using Operators – Rules for Over Loading Operators – Types – Conversions – Inheritance – Extending Classes – Defining Derived Classes – Single, Multi Level Multiple, Hierarchical & Hybrid Inheritance – Virtual Base Classes – Abstract Classes.

UNIT V

Pointers, Virtual Functions & Polymorphism – Pointers to Object - Pointers to Derived Classes – Virtual Functions .Working with Files – Classes for File Stream Operations – Opening and Closing a File – File Pointers & their Manipulations - Sequential I/O Operations.

Suggested Readings:**Text Book:**

1. Balagurusamy, E. (2013). *Object Oriented Programming With C++*, 6th edition, New Delhi: Tata McGraw Hill Publishing Company Ltd.

Reference Books :

1. BjarneStroustrup. (2014). *Programming -- Principles and Practice using C++*, 2nd Edition, Addison-Wesley .
2. BjarneStroustrup,. (2013). *The C++ Programming Language*, 4th Edition, Addison-Wesley.
3. Paul Deitel, Harvey Deitel. (2011). *C++ How to Program*, 8th Edition, Prentice Hall,.
4. D.Ravichandran. (2010). *Programming with C++*. 3rd Edition.. New Delhi: Tata McGraw Hill Publishing Company Ltd.

Website

- W1: <http://www.hscripts.com>
- W2: <http://www3.ntu.edu>
- W3: <http://www.bcanotes.com>
- W4: <http://www.ddegjust.ac.in>



KARPAGAM ACADEMY OF HIGHER EDUCATION
(Deemed University Established Under Section 3 of UGC Act 1956)

Coimbatore - 641021.

(For the candidates admitted from 2016 onwards)

DEPARTMENT OF COMMERCE (CA)

SUBJECT : OBJECT ORIENTED PROGRAMMING WITH C++

SEMESTER : III

SUBJECT CODE: 16CCU302

CLASS : II B.COM CA

UNIT-I

S1 No.	Lecture Duration (Hour)	Topics to be Covered	Support Materials
1	1	PRINCIPLES OF OBJECT ORIENTED PROGRAMMING <ul style="list-style-type: none"> Introduction and Evolution of OOP's 	T:4-7 R2:1-3
2	1	<ul style="list-style-type: none"> Procedure oriented and object oriented paradigm 	
3	1	Basic concepts of OOP's	T:7-15
4	1	Benefits of OOPs <ul style="list-style-type: none"> Objects Classes Data abstraction and Encapsulation Inheritance 	R2:6-11
5	1	Benefits of OOPs <ul style="list-style-type: none"> Polymorphism Dynamic binding Message passing 	
6	1	<ul style="list-style-type: none"> Object Oriented Languages 	W1,W2

7	1	Applications of Oops	W1,W2
8	1	Beginning with C++ <ul style="list-style-type: none">• Introduction to C++	R3:36-55 T:19-30
9	1	<ul style="list-style-type: none">• Applications of C++	R2:13-15
10	1	<ul style="list-style-type: none">• C++ Statements	
11	1	<ul style="list-style-type: none">• Structure of C++ Program	
12	1	Recapitulation	
		Important Questions Discussion	
Total No .Of Hours			12 HOURS

UNIT – II

S1 No.	Lecture Duration (Hour)	Topics to be Covered	Support Materials
1	1	Tokens ,Expressions and Control Structures <ul style="list-style-type: none"> • Tokens, Keywords, Identifiers 	T:35-49
2	1	Basic and User Defined Data Types <ul style="list-style-type: none"> • Variable Declaration • Variable Initialization 	R2:32-38 W3
3	1	Operators <ul style="list-style-type: none"> • Operators in C++ • Operator Overloading 	T:49-64 R2:46-66 W2
4	1	Operators <ul style="list-style-type: none"> • Operator Precedence 	
5	1	Control Structures Decision <ul style="list-style-type: none"> • If & If-Else Statements • Jump Statements 	T:64-69 R2:112-159 R1:124-138
6	1	Control Structures Decision <ul style="list-style-type: none"> • Goto Statements • Break Statements 	
7	1	Looping <ul style="list-style-type: none"> • Switch case Statements • Do-While Statements 	R1:185-210 W1
8	1	For Statements	W3
9	1	Functions in C++ <ul style="list-style-type: none"> • The main Function • Function Prototyping 	T:77-90 R2:179-185
10	1	Functions in C++ <ul style="list-style-type: none"> • Call by Reference, Return by Reference 	R3:273-275
11	1	<ul style="list-style-type: none"> • Inline Functions 	
12	1	Functions in C++	

		<ul style="list-style-type: none">• Function Overloading• Recapitulation Important Questions Discussion	
Total No .Of Hours			12 HOURS

UNIT – III

S1 No.	Lecture Duration (Hour)	Topics to be Covered	Support Materials
1	1	CLASSES AND OBJECTS <ul style="list-style-type: none"> • Introduction • Specifying a Class 	T:96-119
2	1	<ul style="list-style-type: none"> • Defining Member Functions 	
3	1	<ul style="list-style-type: none"> • Nesting of Member Functions 	R2:298-423 W2
4	1	<ul style="list-style-type: none"> • Arrays within a Class 	W3 R3:326-362 W2
5	1	<ul style="list-style-type: none"> • Static Data Members • Static Member Function 	
6	1	<ul style="list-style-type: none"> • Private Member Functions 	
7	1	<ul style="list-style-type: none"> • Array of Objects • Objects as Function Arguments 	T:119-135
8	1	<ul style="list-style-type: none"> • Friendly Functions • Pointers to Members 	R!:185-210 W1
9	1	Constructors and Destructors <ul style="list-style-type: none"> • Constructors 	T:144-164
10	1	<ul style="list-style-type: none"> • Copy Constructor • Dynamic Constructor 	R2:455-476 W4
11	1	<ul style="list-style-type: none"> • Constructor two-dimensional Arrays 	R3:499-502
12	1	<ul style="list-style-type: none"> • Destructor Recapitulation Important Questions Discussion	
Total No .Of Hours			12 HOURS

UNIT – IV

S1 No.	Lecture Duration (Hour)	Topics to be Covered	Support Materials
1	1	Operator overloading	T:171-186 R3:571-669
		<ul style="list-style-type: none">Type Conversion –Introduction	
2	1	<ul style="list-style-type: none">Defining Operator Overloading	W2
3	1	<ul style="list-style-type: none">Overloading unary and binary operatorOverloading binary operator using friends	
4	1	<ul style="list-style-type: none">Manipulation String using Operators	T:186-195 R2:518-524 W3
5	1	<ul style="list-style-type: none">Rules for Operator OverloadingType Conversions	
6	1	Inheritance: <ul style="list-style-type: none">Extending ClassesDefining Derived Classes	T:202-232
7	1	<ul style="list-style-type: none">Single Inheritance	R2:538-548
8	1	<ul style="list-style-type: none">Multilevel Inheritance	
9	1	<ul style="list-style-type: none">Multiple InheritanceHierarchical Inheritance	W3 W4
10	1	<ul style="list-style-type: none">Hybrid Inheritance	
11	1	<ul style="list-style-type: none">Virtual Base ClassesAbstract Classes	R3:708 W4
12	1	Recapitulation	
		Important Questions Discussion	
Total No .Of Hours			12 HOURS

UNIT – V

S1 No.	Lecture Duration (Hour)	Topics to be Covered	Support Materials
1	1	Pointers, Virtual Functions and Polymorphism • Pointers- Introduction	T:251-343
2	1	Pointers to Objects	
3	1	• Pointers to Derived Classes	
4	1	• Virtual Functions	
5	1	Working with Files • Introduction • Classes for File Stream Operations	R3:841-856
6	1	Opening and Closing File	R2: 638-645 W2
7	1	• File Pointers	
8	1	• File Pointers Manipulations	
9	1	• Sequential I/O Operations	W3
10	1	Recapitulation Important Questions Discussion ESE Question Paper Discussions	
11	1	Previous ESE Question Paper Discussions	
12	1	Previous ESE Question Paper Discussions	
Total No .Of Hours			12 HOURS

Suggested Readings:**TEXT BOOK**

1. Balagurusamy, E. (2013). *Object Oriented Programming With C++*, 6th edition, New Delhi: Tata McGraw Hill Publishing Company Ltd.

REFERENCE BOOKS

1. BjarneStroustrup. (2014). *Programming -- Principles and Practice using C++*, 2nd Edition, Addison-Wesley.
2. BjarneStroustrup,. (2013). *The C++ Programming Language*, 4th Edition, Addison- Wesley.
3. Paul Deitel, Harvey Deitel. (2011). *C++ How to Program*, 8th Edition, Prentice Hall.
4. D.Ravichandran. (2010). *Programming with C++*.3rd Edition.. New Delhi: Tata McGraw Hill Publishing Company Ltd.

Website

- W1: <http://www.hscripts.com>
- W2: <http://www3.ntu.edu>
- W3: <http://www.bcanotes.com>
- W4: <http://www.ddegjust.ac.in>



KARPAGAM ACADEMY OF HIGHER EDUCATION

(Deemed University Established Under Section 3 of UGC Act 1956)

Coimbatore - 641021.

(For the candidates admitted from 2016 onwards)

DEPARTMENT OF COMMERCE (CA)

SUBJECT : OBJECT ORIENTED PROGRAMMING WITH C++

SEMESTER : III

SUBJECT CODE: 16CCU302

CLASS : II B.COM CA

UNIT I

Principles of Object- Oriented Programming – A Look at Procedure and Object - Oriented Paradigm – Basic Concepts of Object – Oriented Programming – Benefits of Oop – Object-Oriented Languages – Applications of Oop . Beginning with C++ - What is C++? – Applications of C++ - C++ Statements – Structure of C++ Program.

Unit 1

Introduction

Object-Oriented Programming (OOP) is the term used to describe a programming approach based on **objects** and **classes**. The object-oriented paradigm allows us to organize software as a collection of objects that consist of both data and behaviour. This is in contrast to conventional functional programming practice that only loosely connects data and behaviour.

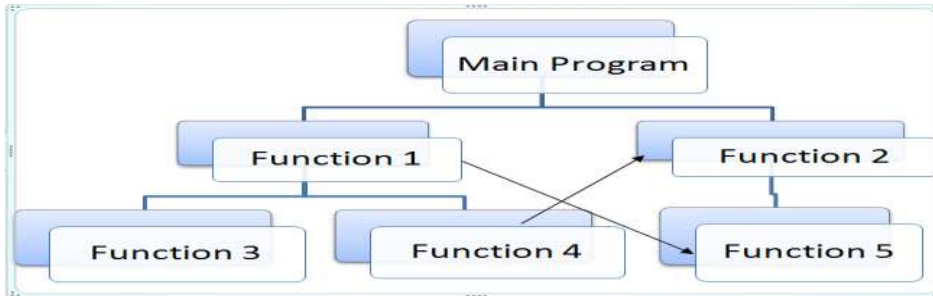
The object-oriented programming approach encourages:

- Modularization: where the application can be decomposed into modules.
- Software re-use: where an application can be composed from existing and new modules.

Procedure-Oriented Programming

High level languages such as COBOL, FORTRAN and C, is commonly known as procedure oriented programming (POP). In the procedure oriented programming, program is divided into sub programs or modules and then assembled to form a complete program. These modules are called functions.

In a multi-function program, many important data items are placed as global so that they may be accessed by all functions. Each function may have its own local data. If a function made any changes to global data, these changes will reflect in other functions. Global data are more unsafe to an accidental change by a function. In a large program it is very difficult to identify what data is used by which function.



Characteristics by procedure-oriented programming are:

Emphasis is on doing things (algorithms).

Large programs are divided into smaller programs known as functions.

Most of the functions share global data.

Data move openly around the system from function to function.

Functions transform data from one form to another.

Employs top-down approach in program design.

Features of POP

1. Mainly focused on writing the algorithms.
2. Most function uses Global data for sharing which are accessed freely from function to function in the system.
3. POP follows the top down approach in program design.
4. It does not have data hiding options.
5. Functions transform data from one form to another.
6. Data can be moved openly from one function to another around the system.
7. Sub-division of large program into smaller programs called functions.
8. Overloading process is not applicable in POP.

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

	Procedure Oriented Programming	Object Oriented Programming
Divided Into	In POP, program is divided into small parts called functions .	In OOP, program is divided into parts called objects .
Importance	In POP, Importance is not given to data but to functions as well as sequence of actions to be done.	In OOP, Importance is given to the data rather than procedures or functions because it works as a real world .
Approach	POP follows Top Down approach .	OOP follows Bottom Up approach .
Access Specifiers	POP does not have any access specifier.	OOP has access specifiers named Public, Private, Protected, etc.
Data Moving	In POP, Data can move freely from function to function in the system.	In OOP, objects can move and communicate with each other through member functions.
Expansion	To add new data and function in POP is not so easy.	OOP provides an easy way to add new data and function.
Data Access	In POP, Most function uses Global data for sharing that can be accessed freely from function to function in the system.	In OOP, data cannot move easily from function to function, it can be kept public or private so we can control the access of data.
Data Hiding	POP does not have any proper way for hiding data so it is less secure .	OOP provides Data Hiding so provides more security .
Overloading	In POP, Overloading is not possible.	In OOP, overloading is possible in the form of Function Overloading and Operator Overloading.
Examples	Example of POP are : C, VB, FORTRAN, Pascal.	Example of OOP are : C++, JAVA, VB.NET, C#.NET.

Principles of Object Oriented Programming: Basic concepts of Object Oriented Programming

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

Objects:

- An object can be considered a "thing" that can perform a set of related activities.
- The set of activities that the object performs defines the object's behavior.
- For example, the hand can grip something or a Student (object) can give the name or address.
- Objects are run time entity or real world entity.

Classes:

- A class is simply a representation of a type of object.
- It is the blueprint/ plan/ template that describe the details of an object.
- A class is the blueprint from which the individual objects are created.
- Class is composed of three things: a name, attributes, and operations.
- For example Student is an object has name, age, course, etc as attributes. Read, write, etc as operations

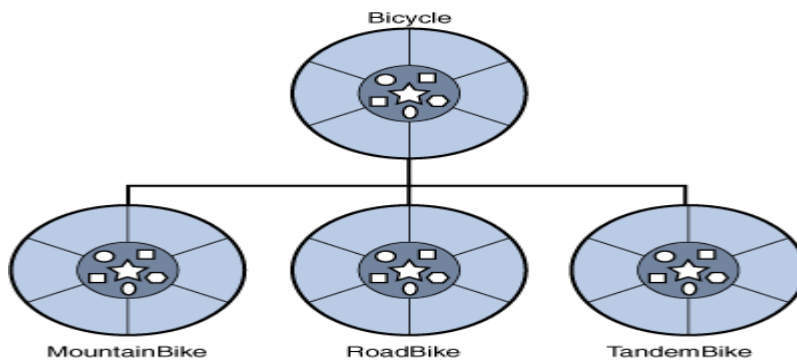
Data abstraction and Encapsulation

- The encapsulation is the inclusion within a program object of all the resources need for the object to function - basically, the methods and the data.
- In OOP the encapsulation is mainly achieved by creating classes, the classes expose public methods and properties.
- The class is kind of a container or capsule or a cell, which encapsulate the set of methods, attribute and properties to provide its indented functionalities to other classes.
- In that sense, encapsulation also allows a class to change its internal implementation without hurting the overall functioning of the system.
- That idea of encapsulation is to hide how a class does it but to allow requesting what to do.
- Abstraction is an emphasis on the idea, qualities and properties rather than the particulars.
- The importance of abstraction is derived from its ability to hide irrelevant details and from the use of names to reference objects.
- Abstraction is essential in the construction of programs.
- It places the emphasis on what an object is or does rather than how it is represented or how it works.

Inheritance

- Ability of a new class to be created, from an existing class by extending it, is called inheritance.
- Different kinds of objects often have a certain amount in common with each other.

- Object-oriented programming allows classes to inherit commonly used state and behavior from other classes.
- In this example, Bicycle now becomes the super class of MountainBike, RoadBike, and TandemBike. In the Java programming language, each class is allowed to have one direct super class, and each super class has the potential for an unlimited number of subclasses:



- The new class created is called as derived class
- The existing class is called as base class.
- The base class provides the property the derived class receives the property.
- It reduces the complexity of the programming.
- This is the most common and most natural and widely accepted way of implement this relationship.

Polymorphism

- Polymorphism is the process taking more then one form.
- More precisely Polymorphisms mean the ability to request that the same operations be performed by a wide range of different types of things.

- In OOP the polymorphisms is achieved by using many different techniques named method overloading, operator overloading and method overriding,
- The method overloading is the ability to define several methods all with the same name.
- The operator overloading (less commonly known as ad-hoc polymorphisms) is a specific case of polymorphisms in which some or all of operators like +, - or == are treated as polymorphic functions and as such have different behaviors depending on the types of its arguments.

Dynamic binding

- Dynamic binding is the process of resolving the function to be associated with the respective functions calls during their runtime rather than compile time.

Message passing

- Every data in an object in oops that is capable of processing request known as message.
- All objects can communicate with each other by sending message to each other
- Message passing, also known as interfacing, describes the communication between objects using their public interfaces.

Benefits of OOP

1. **Reusability:** In OOP's programs functions and modules that are written by a user can be reused by other users without any modification.
2. **Inheritance:** Through this we can eliminate redundant code and extend the use of existing classes.
3. **Data Hiding:** The programmer can hide the data and functions in a class from other classes. It helps the programmer to build the secure programs.

4. **Reduced complexity of a problem:** The given problem can be viewed as a collection of different objects. Each object is responsible for a specific task. The problem is solved by interfacing the objects. This technique reduces the complexity of the program design.
5. **Easy to maintain and Upgrade:** OOP makes it easy to maintain and modify existing code as new objects can be created with small differences to existing ones.
6. **Message Passing:** The technique of message communication between objects makes the interface with external systems easier.
7. **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.

Object Oriented Language

Object-oriented programming is not the right of any particular languages.

Like structured programming, OOP concepts can be implemented using languages such as C and Pascal. It is specially designed to support the OOP concepts makes it easier to implement them.

Object Oriented Languages has two categories:

1. **Object-based programming languages**
2. **Object-oriented programming languages**

Object-based programming is the style of programming that primarily supports encapsulation and object identity.

Major feature that are required for object based programming are:

- Data encapsulation
- Data hiding and access mechanisms
- Automatic initialization and clear-up of objects
- Operator overloading Languages that support programming with objects are said to be objects-based programming languages. They do not support

inheritance and dynamic binding. Ada is a typical object-based programming language.

Object-Oriented programming language incorporates all of object-based programming features along with two additional features, namely, inheritance and dynamic binding.

Object-oriented programming can therefore be characterized by the following statements:

Object-based features + inheritance + dynamic binding

Applications of OOPs

- For Develop Graphical related application like computer and mobile games.
- To evaluate any kind of mathematical equation use C++ language.
- C++ Language are also used for design OS. Like window xp.
- Google also use C++ for Indexing
- Few parts of apple OS X are written in C++ programming language.
- Internet browser Firefox are written in C++ programming language
- All major applications of adobe systems are developed in C++ programming language. Like Photoshop, ImageReady, Illustrator and Adobe Premier.
- Some of the Google applications are also written in C++, including Google file system and Google Chromium.
- C++ are used for design database like MySQL.

Beginning with C++

What is C++: C++ is a general-purpose object-oriented programming (OOP) language, developed by Bjarne Stroustrup, and is an extension of the C language. It is therefore possible to code C++ in a "C style" or "object-oriented style". C++ is one of the most versatile languages in the world. It is used nearly

everywhere for everything... systems programming (operating systems, device drivers, database engines, embedded, Internet of Things, etc.)

Applications of C++

- Client-Server System
- Object Oriented Database
- Object Oriented Distributed Database
- Real Time Systems Design
- Simulation and Modeling System
- Hypertext, Hypermedia
- Neural Networking and Parallel Programming
- Decision Support and Office Automation Systems
- CIM/CAD/CAM Systems
- AI and Expert Systems

C++ Statements

Preprocessor directives

A preprocessor directive begins with the character #. This must either be the first character of the line or the first character of the line after some leading whitespace.

Comments

Comments may be of the form:

`/ / Comment \n`

(Or)

`/* comment */`

- The first form allows a trailing comment on a single line, while the second form allows comments that span multiple lines.
- Comments may appear anywhere.

Declarations

- Declarations give the compiler information about the types, storage requirements and initial values of identifiers.
- **General form:** void type identifier initializer;

Function Declarations

```
void type identifier (formal_argument_list )  
{  
    function_body  
}  
Executable statements
```

while (expression) statement

do statement while (expression)

for (expression_1; expression_2; expression_3) statement

equivalent to:

expression_1; while (expression_2) { statement expression_3 }

switch (expression)

{

declarations

case constant_expression: statement

...

default: statement

}

break;

continue;

return;

return expression;

goto statement_label;

statement_label: executable_statement

Structure of C++ program

- Include section
- Class declaration
- Member function definition
- Main function program

Output Operator:

cout<<

■ **Syntax:**

cout<<argument;

■ **Example:**

cout<<“Welcome to C++”;

Program:

```
#include<iostream.h>
void main()
{
    cout<<“Welcome to C++”;
}
```

Compiling and Linking:

- Create the source code.
- Save the code with extension .cpp

- Compile the program with
Alt + F9 (or)
Select Compile option from Compile menu

- Run the program with
Ctrl + F9 (or)
Select Run option from Run menu

Input Operator:

Cin>>

- **Syntax:**
cin<<argument;

- **Example:**
cin>>a;
a is a variable.

Program:

```
#include<iostream.h>
void main()
{
    float n1,n2,sum,avg;
    cout<<"Enter the two No:";
    cin>>n1>>n2;
    sum=n1+n2;
    avg=sum/2;
    cout<<"Sum:"<<sum;
    cout<<"\nAverage:"<<avg;
}
```



KARPAGAM ACADEMY OF HIGHER EDUCATION
(Deemed University Established Under Section 3 of UGC Act
1956)
Coimbatore – 21
(For the candidates admitted from 2016 onwards)
DEPARTMENT OF COMMERCE (CA)

SUBJECT : OBJECT ORIENTED PROGRAMMING WITH C++
SEMESTER : III

SUBJECT CODE: 16CCU302 **CLASS : II B.COM CA**

POSSIBLE QUESTIONS – UNIT I

PART A (1 Mark)

(Online Examination)

PART B (2 Marks)

1. Define Objects
2. What is C++ Statements
3. List out the Basic Concepts of Oops
4. Specify object Oriented Languages
5. List out Applications of Oops
6. List out Applications of OOPs
7. What is Structure of C++ Program
8. Differentiate between POP and OOPs
9. What is Procedure Oriented Programming
10. What is Object Oriented Programming
11. What is C++ statements
12. What is data abstraction
13. Write any four features of OOPS

PART C (6 Marks)

1. Describe about Procedure Oriented Programming
2. Explain Object Oriented Programming
3. Write a Program for Structure of C++ Program
4. List out Benefits of OOPS
5. Explain in detail about basic concepts of Object Oriented Programming.
6. Write a program to find average of two numbers
7. Differentiate between POP and OOPS.
8. Enumerate History of C++
9. Write a program to find largest of three given numbers.
10. Write in about object oriented languages



KARPAGAM ACADEMY OF HIGHER EDUCATION

(Deemed University Established Under Section 3 of UGC Act 1956)

Coimbatore - 641021.

(For the candidates admitted from 2016 onwards)

DEPARTMENT OF COMMERCE (CA)

SUBJECT : OBJECT ORIENTED PROGRAMMING WITH C++

SEMESTER : III

SUBJECT CODE: 16CCU302

CLASS : II B.COM CA

UNIT II

Tokens, Expressions and Control Structures – Tokens – Keywords – Identifiers – Basic & User – Defined Data Types – Operators in C++ - Operator Over Loading – Operator Precedence – Control Structures – Functions in C++ - The Main Function – Function Prototyping – Call By Reference – Return By Reference – In Line Functions – Function Over Loading – Friend and Virtual Functions.

Unit – II

Tokens:

Smallest unit of a Program is called Token.

- Keywords
- Identifiers
- Constants
- Strings
- Operators
- Special Symbols

Keywords

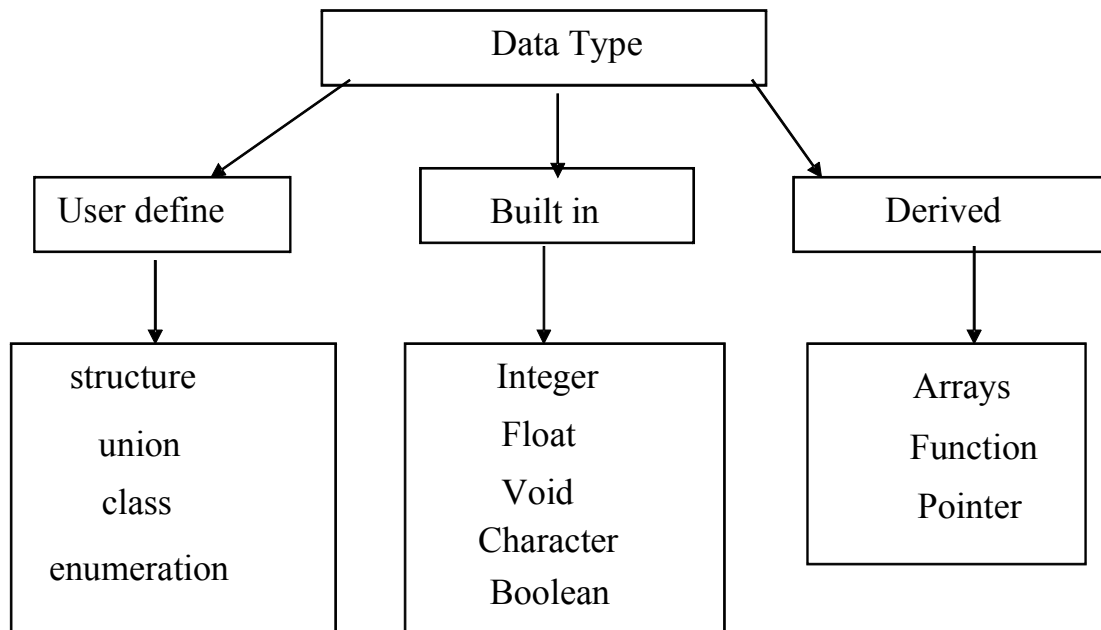
- Keywords are reserved words.
- Has its predefine meaning.
- C++ consist of c keywords and additional keywords of its own

Keyword List:

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	
----------	---------	-------	--

Data Types:



Integer Type : Integer data type are like whole numbers, they also include negative numbers but does not support decimal numbers.

Type	Storage size	Value range
int	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295

short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
long	4 bytes	-2,147,483,648 to 2,147,483,647
unsigned long	4 bytes	0 to 4,294,967,295

Float-point Type : Float data type allows user to store decimal values in a variable.

Type	Storage size	Value range	Precision
float	4 byte	1.2E-38 to 3.4E+38	6 decimal places
double	8 byte	2.3E-308 to 1.7E+308	15 decimal places
long double	10 byte	3.4E-4932 to 1.1E+4932	19 decimal places

Character Type : Character data type is used to store only one letter, digit, symbol at a time.

Type	Storage size	Value range
char	1 byte	-128 to 127 or 0 to 255

unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127

Array : Array is a collection of similar data type. A single variable can hold only one value at a time, If we want a variable to store more than one value of same type we use array.

Pointers : A normal variable is used to store value. A pointer variable is used to store address / reference of another variable.

Variables

Variables are used to store values. variable name is the name of memory location where value is stored. It must be alphanumeric, only underscore is allowed in a variable name. It is composed of letters, digits and only underscore. It must begin with alphabet or underscore. It cannot be begin with numeric.

Declaration of Variable

Declaration will allocate memory for specified variable with garbage value.

Syntax :

```
Data-Type Variable-name;
```

Examples :

```
int a;
```



```
float b;  
  
char c;
```

Variable (Identifiers):

- Identifiers are user define name space
- It change its value during the execution of the program
- It refers the names of variables, functions, arrays, classes, etc

Rules:

1. Only alphabetic characters, digits and underscores are permitted.
2. Cannot start with digits.
3. Uppercase and lowercase are distinct
4. Keyword should not be as a variable name.

Example:

A, welcome

Initialization of Variable

Initialization means assigning value to declared variable. Every value will overwrite the previous value.

Examples :

```
a = 10;  
  
b = 4.5;  
  
c = 'a';
```

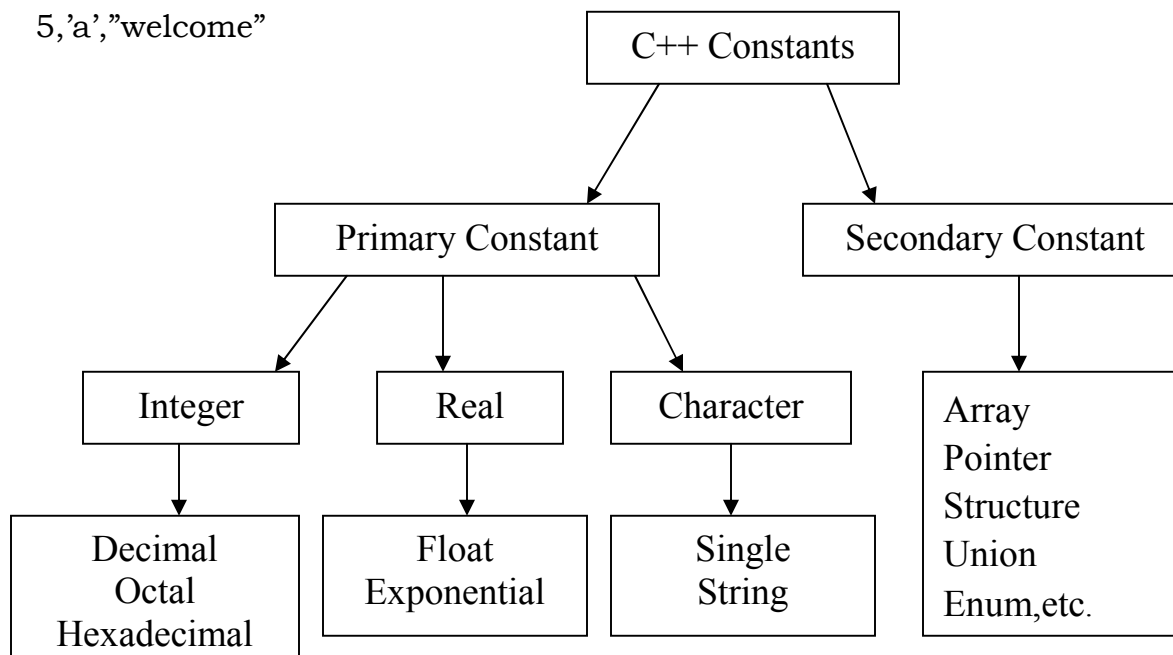
Character value must be enclosed with single quotes.

Constants:

- It does not change its value during the execution of the program

Example:

5,'a',"welcome"



Operators:

- Arithmetic operators
- Assignment operators
- Increment/Decrement
- Comparison operators
- Logical operators
- Bitwise operators

Assignment operator is used to copy value from right to left variable.

Operator	Name	Description	Example

Tokens, Expression and Control Structure 2016 – 2019 Batch

=	Equal sign	Copy value from right to left.	X = Y, Now both X and Y have 5
+=	Plus Equal to	Plus Equal to operator will return the addition of right operand and left operand.	X += Y is similar to X = X + Y, now X is 7
-=	Minus Equal to	Minus Equal to operator will return the subtraction of right operand from left operand.	X -= Y is similar to X = X - Y, now X is 3
*=	Multiply Equal to	Multiply Equal to operator will return the product of right operand and left operand.	X *= Y is similar to X = X * Y, now X is 10
/=	Division Equal to	Division Equal to operator will divide right operand by left operand and return the quotient.	X /= Y is similar to X = X / Y, now X is 2.5
%=	Modulus or Mod Equal to	Modulus Equal to operator will divide right operand by left operand and return the mod (Remainder).	X %= Y is similar to X = X % Y, now X is 1

Arithmetic operators are used for mathematical operations.

Operator	Name	Description	Example
+	Plus	Return the addition of left and right operands.	(X + Y) will return 7
-	Minus	Return the difference b/w right operand from left operand.	(X - Y) will return 3
*	Multiply	Return the product of left and right operands.	(X * Y) will return 10
/	Division	Return the Quetiont from left operand by right operand.	(X / Y) will return 2(both are int, int doesn't support decimal)
%	Modulus or Mod	Return the Modulus (Remainder) from left operand by right operand.	(X % Y) will return 1

Relational operators are used for checking conditions whether the given condition is true or false. If the condition is true, it will return non-zero value, if the condition is false, it will return 0.

Operator	Name	Description	Example
>	Greater then	Check whether the left operand is greater then right operand or not.	(X > Y) will return true
<	Smaller then	Check whether the left operand is smaller then right operand or not.	(X < Y) will return false
>=	Greater then or Equal to	Check whether the left operand is greater or equal to right operand or not.	(X >= Y) will return true
<=	Smaller then or Equal to	Check whether the left operand is smaller or equal to right operand or not.	(X <= Y) will return false
==	Equal to	Check whether the both operands are equal or not.	(X == Y) will return false
!=	Not Equal to	Check whether the both operands are equal or not.	(X != Y) will return true

Logical operators are used in situation when we have more then one condition in a single if statement.

Operator	Name	Description	Example
&&	AND	Return true if all conditions are true, return false if any of the condition is false.	if(X > Y && Y < X) will return true
	OR	Return false if all conditions are false, return true if any of the condition is true.	if(X > Y X < Y) will return true
!	NOT	Return true if condition is false, return false if condition is true.	if(!(X>y)) will return false

The conditional operator is also known as **ternary operator**. It is called ternary operator because it takes three arguments. First is condition, second and third is value. The conditional operator check the condition, if condition is true, it will return second value, if condition is false, it will return third value.

Syntax :

```
val = condition ? val1 : val2;
```

Example :

```
void main()
{
```

```
int X=5,Y=2,lrg;

lrg = (X>Y) ? X : Y;

cout << "\nLargest number is : " << lrg;

}
```

Output :

Largest number is : 5

Binary operators are those operators that works with at least two operands such as (Arithmetic operators) +, -, *, /, %.

Unary operators are those operators that works with singal operands such as (Increment or Decrement operators) ++ and --.

Special Operators:

- ::- Scope resolution operator
- >>-Insertion Operator
- <<-Extraction Operator
- ::*- Pointer-to-member decelerator
- ->*- Pointer-to-member operator

- .*- Pointer to member operator
- new- Memory management operator
- delete- Memory release operator
- endl- Line feed operator
- sew- Memory allocation operator
- setw- Field width operator

Scope Resolution Operator:

:: - Used to access values or methods.

Syntax:

::variable name;
:: function name;

Examples:

::a;
::add();

Program:

```
#include<iostream.h>
int m=10;
void main()
{
    int m=20;
    {
        int m=40;
        cout<<"Value of m in inner block:"<<m;
        cout<<"Value of m in outter block:"<<::m;
    }
    cout<<"Value of m in inner block:"<<m;
    cout<<"Value of m in outter block:"<<::m;
}
```


Manipulator:

endl:

used instead of "\n".

Example:

```
cout<<"welcome to csc"<<endl;
```

setw:

used for neat alignment during display.

Syntax:

```
setw (int value)
```

Example:

```
cout<<setw(5);
```

Program:

```
#include<iostream.h>
#include<iomanip.h>
#include<conio.h>
void main()
{
    int m1=50,m2=500,m3=5000;
    clrscr();
    cout<<setw(5)<<"m1:"<<setw(5)<<m1<<endl;
    cout<<setw(5)<<"m2:"<<setw(5)<<m2<<endl;
    cout<<setw(5)<<"m3:"<<setw(5)<<m3<<endl;
}
```

Output:

```
m1:  50
m2: 500
m3: 5000
```

Type Cast:

- Convert the data type of a variable to some other data type variable

Syntax:

(type name) expression //c

type name (expression) //c++

Example:

float (i);

Decision making statements

if ..else, jump, goto, break, continue- switch case statements

Control Structure:

- Sequence structure
- Selection structure
- Loop structure

Selection Structure:

- if statement
- if...else
- Nested if
- if... else ladder
- switch statement

if Statement:

if statement takes condition in parenthesis and a block of statements within braces. If condition is true, it will return non-zero value, and statements given in if block will get execute.

Syntax:

if(condition)

{

True Block;

}

Next Statement;

Example:

```
if(i>10)
    cout<<"i greater than 10";
```

if ...else Statement

if statement takes condition in parenthesis and a block of statements within braces. If condition is true, it will return non-zero value, and statements given in if block will get execute. If condition is false, it will returns zero, and statements given in else block will get execute.

Syntax:

```
    if(condition)
    {
        True Block;
    }
    else
    {
        False Block;
    }
    Next Statement;
```

Example:

```
if(i>10)
    cout<<"i greater than 10";
else
    cout<<"i less than 10";
```

Nested if Statement:

In nested if-else, one if-else statement contains another if-else statement.

Syntax:

```
    if(condition 1)
    {
```

```
        if(condition 2)
        {
            True block
        }
        else
        {
            False block condition 2;
        }
    }
    else
    {
        False block condition 1;
    }
    Next Statement;
```

Example:

```
if(m1>40)
{
    if(m2>40)
    {
        cout<<"pass";
    }
    else
    {
        cout<<"Fail";
    }
}
else
{
    cout<<"Fail";
}
```

}

if... else ladder:

if-else ladder is used for checking multiple conditions, if the first condition will not satisfy, compiler will jump to else block and check the next condition, whether it is true or not and so on.

Syntax:

```
if(condition 1)
{
    True block-1
}
else if(condition 2)
{
    True block -2;
}
else
{
    False block;
}
Next Statement;
```

Example:

```
if(a>b)
{
    if(a>c)
    {
        cout<<"A is Greatest";
    }
    else
    {
        cout<<"C is Greatest";
    }
}
```

```
        }  
    }  
    else  
    if(b>c)  
    {  
        cout<<"B is Greatest";  
    }  
    else  
    {  
        cout<<"C is Greatest";  
    }  
}
```

Switch Case:

Syntax:

```
switch(expression)  
{  
    case exp1:  
        Statements  
    case exp2:  
        Statements  
    .....  
    .....  
    default:  
        Statements  
}
```

Example

```
i=4;  
switch(i)  
{  
    case 1:
```

```
        cout<<"one";  
case 2:  
        cout<<"two";  
case 3:  
        cout<<"three";  
default:  
        cout<<"Wrong Choice";  
}
```

do-while – while statement, for statement

Loop Structure

Entry control:

Entry control Structure checks the condition First and the Statement is executed

- while loop
- for loop

Exit control:

Exit control Structure First the Statement is Executed and then checks the condition

- do... while

While Loop:

While loop is also called entry control loop because, in while loop, compiler will 1st check the condition, whether it is true or false, if condition is true then execute the statements.

Syntax:

```
while(Condition)
{
    Statement Block
}
```

Example:

```
while(i<5)
{
    cout<<"Welcome";
    i++;
}
```


For Loop:

In for loop has initialization, condition and increment/decrement all together. Initialization will be done once at the beginning of loop. Then, the condition is checked by the compiler. If the condition is false, for loop is terminated. But, if condition is true then, the statements are executed until condition is false.

Syntax:

```
for(initialization ; condition checking ; Increment/Decrement)
{
    Statement Block
}
```

Example:

```
for(i=1;i<5;i++)
{
    cout<<"Welcome";
}
```

Do..While

The do-while loop is also called exit control loop because, in do-while loop, compiler will 1st execute the statements, then check the condition, whether it is true or false.

Syntax:

```
do
{
}while(condition);
```

Example:

```
do
{
    cout<<"Welcome";
    i++;
}while(i<5);
```

Difference b/w while loop and do-while loop

while loop	do-while loop
It is entry control loop.	It is exit control loop.
In this loop condition is checked before loop execution.	In this loop condition is checked at the end of loop.
It will never execute loop if condition is false.	It will executes loop at least once when the initial condition is false.
There is no semicolon at the end of while statement	There is semicolon at the end of while statement.

Jump Statements in C++

Jump statements are used to interrupt the normal flow of program.

Types of Jump Statements

- Break
- Continue
- GoTo

Break Statement

The break statement is used inside loop or switch statement. When compiler finds the break statement inside a loop, compiler will abort the loop and continue to execute statements followed by loop.

Example of break statement

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

```
void main()
{
    int a=1;

    while(a<=10)
    {
        if(a==5)
            break;

        cout << "\nStatement " << a;
        a++;
    }
    cout << "\nEnd of Program.";
}
```

Output :

```
Statement 1
Statement 2
Statement 3
Statement 4
End of Program.
```

Continue Statement

The continue statement is also used inside loop. When compiler finds the break statement inside a loop, compiler will skip all the following statements in the loop and resume the loop.

Example of continue statement

```
#include<iostream.h>

void main()
{
    int a=0;

    while(a<10)
    {

        a++;

        if(a==5)
            continue;

        cout << "\nStatement " << a;

    }
    cout << "\nEnd of Program.";
}
```

Output :

```
Statement 1
Statement 2
Statement 3
Statement 4
```

```
Statement 6
Statement 7
Statement 8
Statement 9
Statement 10
End of Program.
```

Goto Statement

The goto statement is a jump statement which jumps from one point to another point within a function.

Syntax of goto statement

```
goto label;
-----
-----
label:
-----
-----
```

In the above syntax, label is an identifier. When, the control of program reaches to goto statement, the control of the program will jump to the label: and executes the code after it.

Example of goto statement

```
#include<iostream.h>

void main()
```

```
{  
  
    cout << "\nStatement 1.";   
    cout << "\nStatement 2.";   
    cout << "\nStatement 3.";   
  
    goto last;   
  
    cout << "\nStatement 4.";   
    cout << "\nStatement 5.";   
  
    last:   
  
    cout << "\nEnd of Program.";   
}
```

Output :

```
Statement 1.  
Statement 2.  
Statement 3.  
End of Program.
```

Function in C++

A function is a block of codes that performs a specific task and may return value. The main() function is the first user defined function invoked by the compiler. While it is possible to write any code within main function, it leads number of problems. The program may become too large and complex and it is difficult to test, debug and maintain the complex code. For that reason, We use function to place independent code in separate modules called function or

subprogram. In order to make a program using function, we need to perform the following three steps.

- Function declaration
- Function definition
- Function call

Function declaration

Like variables, all the functions must be declared. Function declaration statement includes function name, what function will take and what function will return.

Syntax :

```
return-type function-name(argument list);
```

return-type : type of value function will return.

function-name : any valid C++ identifier.

argument list : represents the type and number of value function will take, values are sent by the calling statement.

Example for declaration of function

If we want to return the sum of two integer numbers and function will take two numbers as argument then the function declaration statement will be:

```
int Add(int, int);
```

Function definition

Function definition includes the actual working or implementation.

Syntax for defining function

```
return-type function-name(argument list)
{
    -----
    body of function
    -----
}
```

The body of function contains the number of statements to perform specific task.

Example for definition of function

The body of function for calculating sum of two integer numbers.

```
int Add(int x,int y)
{
    int sum;

    sum = x + y;

    return sum;
}
```

Function call or Function invoke

To execute function we must call it. A function can be called or invoked by using function name followed by list of arguments (values) that function definition will receive to perform task.

Syntax for calling or invoke function


```
var = function-name(val1,val2...n);
```

var can be any variable that will receive value returning from function definition.

Example for calling or invoke function

Considering the above example, function calling statement should be :

```
int rs;  
rs = Add(10,20);    //calling statement  
cout << "\nThe sum is : " << rs;
```

Passing argument to a function

Like normal variable, pointer variable can be passed as function argument and it can return from function.

There are two approaches to passing argument to a function:

- Call by Value
- Call by Reference/Address

Call by Value

In this approach, the values are passed as function argument to the definition of function.

Example of call by value

```
#include<iostream.h>  
  
void main()  
{  
    int A=10,B=20;
```

```
    cout << "\nValues before calling";
    cout << "\nA : " << A;
    cout << "\nB : " << B;

    fun(A,B);                //Statement    1

    cout << "\nValues after calling";
    cout << "\nA : " << A;
    cout << "\nB : " << B;

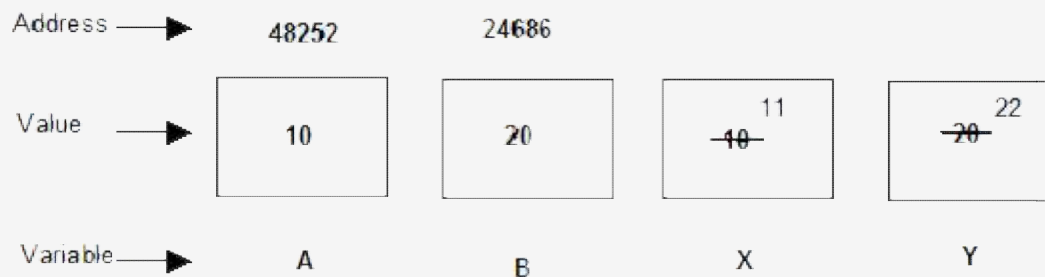
}

void fun(int X,int Y)        //Statement    2
{
    X=11;
    Y=22;
}
```

Output :

```
Values before calling
A : 10
B : 20
Values after calling
A : 10
B : 20
```

In the above example, statement 1 is passing the values of A and B to the calling function fun(). fun() will receive the value of A and B and put it into X and Y respectively. X and Y are value type variables and are local to fun(). Any changes made by value type variables X and Y will not effect the values of A and B.



Call by Reference

In this approach, the references/addresses are passed as function argument to the definition of function.

Example of call by reference

```
#include<iostream.h>

void main()
{
    int A=10,B=20;
```

```
cout << "\nValues before calling";
cout << "\nA : " << A;
cout << "\nB : " << B;

fun(&A,&B);                //Statement    1

cout << "\nValues after calling";
cout << "\nA : " << A;
cout << "\nB : " << B;

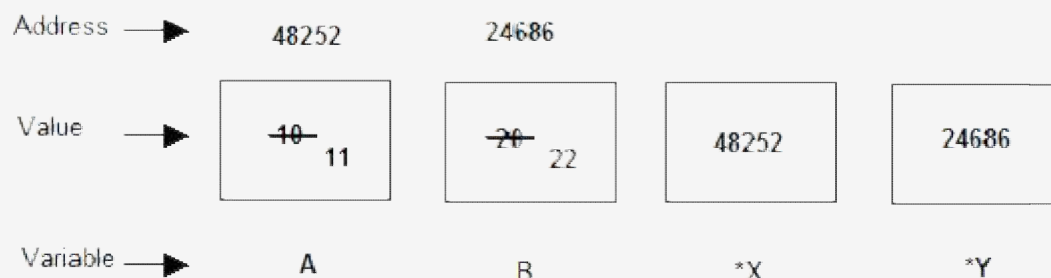
}

void fun(int *X,int *Y)      //Statement    2
{
    *X=11;
    *Y=22;
}
```

Output :

```
Values before calling
A : 10
B : 20
Values after calling
A : 11
B : 22
```

In the above example, statement 1 is passing the reference of A and B to the calling function fun(). fun() must have pointer formal arguments to receive the reference of A and B. In statement 2 *X and *Y is receiving the reference A and B. *X and *Y are reference type variables and are local to fun(). Any changes made by reference type variables *X and *Y will change the values of A and B respectively.



Difference between Call by Value and Call by Reference.

Call by Value	Call by Reference
The actual arguments can be variable or constant.	The actual arguments can only be variable.
The values of actual argument are sent to formal argument which are normal variables.	The reference of actual argument are sent to formal argument which are pointer variables.
Any changes made by formal arguments will not reflect to actual	Any changes made by formal arguments will reflect to actual

arguments.

arguments.

Inline functions

One of the advantages of using function is to save memory space by making common block for the code we need to execute many times. When compiler invoke / call a function, it takes extra time to execute such as jumping to the function definition, saving registers, passing value to argument and returning value to calling function. This extra time can be avoidable for large functions but for small functions we use inline function to save extra time.

When we make an inline function, compiler will replace all the calling statements with the function definition at run-time.

- Expand itself code during the execution of the program.
- Keyword: **inline**

Syntax:

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

Example of inline function

```
#include<iostream.h>
#include<conio.h>
inline int add(int a,int b)
{
    return(a+b);
}
```

```
void main()
{
    int m1,m2;
    clrscr();
    cout<<"Enter the first number:";
    cin>>m1;
    cout<<"Enter the Second number:";
    cin>>m2;
    cout<<"Addition Result:"<<add(m1,m2)<<endl;
}
```

Function overloading

More than one function with same name, with different signature in a class or in a same scope is called function overloading. Signature of function includes:

- Number of arguments
- Type of arguments
- Sequence of arguments

Example of function overloading

```
#include<iostream.h>

#include<conio.h>

class CalculateArea
{
```

```
public:

void Area(int r)          //Overloaded Function 1
{
    cout<<"\n\tArea of Circle is : "<<3.14*r*r;
}

void Area(int l,int b)    //Overloaded Function 2
{
    cout<<"\n\tArea of Rectangle is : "<<l*b;
}

void Area(float l,int b)  //Overloaded Function 3
{
    cout<<"\n\tArea of Rectangle is : "<<l*b;
}

void Area(int l,float b)  //Overloaded Function 4
{
    cout<<"\n\tArea of Rectangle is : "<<l*b;
}
```



```
};
```

```
void main()
```

```
{
```

```
    CalculateArea C;
```

```
    C.Area(5);    //Statement 1
```

```
    C.Area(5,3);  //Statement 2
```

```
    C.Area(7,2.1f); //Statement 3
```

```
    C.Area(4.7f,2); //Statement 4
```

```
}
```

Output :

Area of Circle is : 78.5

Area of Rectangle is : 15

Area of Rectangle is : 14.7

Area of Rectangle is : 29.4

Example Program 2:

```
#include<iostream.h>
#include<conio.h>
int volume(int);
double volume(double,int);
long volume(long,int,int);
void main()
{
    clrscr();
    cout<<"Function with one argument:"<<volume(10)<<"\n";
    cout<<"Function with two argument:"<<volume(5.6,10)<<endl;
    cout<<"Function with three argument:"<<volume(561,67,89)<<endl;
}
int volume(int s)
{
    return(s*s*s);
}
double volume(double r,int h)
{
    return(3.14*r*r*h);
}
long volume(long l,int b,int h)
{
    return(l*b*h);
}
```



KARPAGAM ACADEMY OF HIGHER EDUCATION
(Deemed University Established Under Section 3 of UGC Act 1956)
Coimbatore - 641021.

(For the candidates admitted from 2016 onwards)

DEPARTMENT OF COMMERCE (CA)

SUBJECT : OBJECT ORIENTED PROGRAMMING WITH C++

SEMESTER : III

SUBJECT CODE: 16CCU302

CLASS : II B.COM CA

UNIT III



Classes and Objects – Introduction – Specifying a Class – Defining Member Function – Nesting of Member Functions – Private Member Functions – Arrays within a Class- Static Data Members – Static Member Functions – Array of Objects – Objects as Function Arguments – Friendly Functions – Pointers to Members. Constructors & Destructors – Constructors – Copy Constructors – Dynamic Constructors – Construction Two- Dimensional Arrays – Destructors.

Unit – III

Classes and objects : Specifying a class

- Class is composed of three things: a name, attributes, and operations.
- Class is a way to bind the data and its associated functions together

Class specification has 2 parts:

-  Class Declaration.
-  Class function definitions

Access Specifies:

- The Status of the accessibility of the data members are determined by the Access Specifies
- There are 3 access specifies
 - Public
 - Private
 - Protected

Public:

It allows functions and data to be accessible to any part of the program.

Private:

It allows functions and data cannot be accessible to any part of the program except the class where it is declared.

Protected

It allows functions and data to be accessible to only the derived classes.

Class Declaration:

Syntax:

Prepared by Dr.S.Hemalatha, Department of Commerce (Computer Application) ,
KAHE

```
class class_name
{
    private:
        variable declaration;
        function declaration;
    public:
        variable declaration;
        function declaration;
}
```

Example:

```
class book
{
    int pgno;
    public:
        void getpage();
}
```

Creation of Objects:

Once the class is created, one or more objects can be created from the class as objects are instance of the class.

Just as we declare a variable of data type int as:

```
int x;
```

Objects are also declared as:

class_name followed_by object_name;

Example:

```
exforsys e1;
```

This declares e1 to be an object of class exforsys.

Accessing Class Members:

Creating Object:

Syntax:

```
classname object_name;
```

Example:

```
book i;
```

Accessing Methods:

Syntax:



```
object.function_name(argument)
```

Example:

```
i.getpage();
```

Defining member functions

Defining a Member

- Definition in 2 places
-  Outside the class definition.
-  Inside the class definition.

Outside the Class Definition

Syntax:

```
return_type class_name :: function_name(argument list)
{
    -----
    body of function
}
```

Prepared by Dr.S.Hemalatha, Department of Commerce (Computer Application) ,
KAHE

```

        -----
    }

```

return_type Type of value function will return.

class_name:: A program may contain more than one class and these classes may have similar member functions. Class_name:: tells the compiler which class the function belongs to and the scope of the member function is restricted to the class_name.

function_name Can be any valid C++ identifier.

argument list Represents the type and number of value function will take, values are sent by the calling statement.

Example of defining member function outside class

```

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

class Employee
{
    int Id;
    char Name[25];
    int Age;
    long Salary;

    public:
    void GetData();
    void PutData();

```

```
};

void Employee :: GetData()          //Statement 1 : Defining GetData()
{
    cout<<"\n\tEnter Employee Id : ";
    cin>>Id;

    cout<<"\n\tEnter Employee Name : ";
    cin>>Name;

    cout<<"\n\tEnter Employee Age : ";
    cin>>Age;

    cout<<"\n\tEnter Employee Salary : ";
    cin>>Salary;
}

void Employee :: PutData()          //Statement 2 : Defining PutData()
{
    cout<<"\n\nEmployee Id : "<<Id;
    cout<<"\nEmployee Name : "<<Name;
    cout<<"\nEmployee Age : "<<Age;
    cout<<"\nEmployee Salary : "<<Salary;
}

void main()
{
```



```
Employee E;           //Statement 3 : Creating Object

E.GetData();          //Statement 4 : Calling GetData()
E.PutData();           //Statement 5 : Calling PutData()

}
```

Output :

```
Enter Employee Id : 1
Enter Employee Name : Kumar
Enter Employee Age : 29
Enter Employee Salary : 45000
```

```
Employee Id : 1
Employee Name : Kumar
Employee Age : 29
```

```
Employee Salary : 45000
```

Inside the Class Definition:

Syntax:

```
return_type function_name(argument list)
{
    -----
    body of function
    -----
}
```

return_type Type of value function will return.

Prepared by Dr.S.Hemalatha, Department of Commerce (Computer Application) ,
KAHE

function_name Can be any valid C++ identifier.

argument list Represents the type and number of value function will take, values are sent by the calling statement.

Definition of member function inside class is similar to defining normal function. There is no need to tell compiler about the class the function belongs to because the definition of member function is already in the class.

Example of defining member function inside class

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

class Employee
{
    int Id;
    char Name[25];
    int Age;
    long Salary;

    public:
    void GetData()          //Statement 1 : Defining GetData()
    {
        cout<<"\n\tEnter Employee Id : ";
        cin>>Id;

        cout<<"\n\tEnter Employee Name : ";
        cin>>Name;
```

```
        cout<<"\n\tEnter Employee Age : ";
        cin>>Age;

        cout<<"\n\tEnter Employee Salary : ";
        cin>>Salary;
    }

    void PutData()          //Statement 2 : Defining PutData()
    {
        cout<<"\n\nEmployee Id : "<<Id;
        cout<<"\nEmployee Name : "<<Name;
        cout<<"\nEmployee Age : "<<Age;
        cout<<"\nEmployee Salary : "<<Salary;
    }

};

void main()
{

    Employee E;           //Statement 3 : Creating Object

    E.GetData();          //Statement 4 : Calling GetData()
    E.PutData();           //Statement 5 : Calling PutData()

}
```

Output :

```
Enter Employee Id : 1
Enter Employee Name : Kumar
Enter Employee Age : 29
Enter Employee Salary : 45000
```

```
Employee Id : 1
Employee Name : Kumar
Employee Age : 29
```

```
Employee Salary : 45000
```

Static data members

Static Variables (Static Data members):

- By default it is initialized to zero
- Only one copy of that variable is created for entire class
- It is visible only within the class, but lifetime is the entire program
- static is the keyword used to declare static data members

Syntax:

```
static datatype variable=value
```

Example:

```
static int i=5;
```

Program:

```
#include<iostream.h>
void main()
{
    static int i=3;
    while(i)
```

```
{  
    cout<<i<<"\n";  
    i--;  
    main();  
}  
}
```

Static member functions

Static Member Functions:

- A static function can have access to only other static members declared in the class
- A static member function can be called using the class name

Syntax:

classname::functionname

Program:

```
#include<iostream.h>  
class test  
{  
    int code;  
    static int cn;  
public:  
    void set()  
    {  
        code=++cn;  
    }  
    void showcode()  
    {  
        cout<<"\nObject number:"<<code<<"\n";  
    }  
}
```

```
static void showcount()
{
    cout<<"\ncount:"<<cn;
}
};
int test::cn;
void main()
{
    test t1,t2;
    t1.set();
    t2.set();
    test::showcount();
    test t3;
    t3.set();
    test::showcount();
    t1.showcode();
    t2.showcode();
    t3.showcode();
}
```

Array of objects

- An object of class represents a single record in memory,
- an array is a collection of similar type, therefore an array can be a collection of class type..

Syntax:

Class name object[size];

Example:

Item i[50];

Program:

```
#include<iostream.h>
class employee
{
    char name[30];
    float age;
public:
    void get();
    void put();
};

void employee::get()
{
    cout<<"Enter name:";
    cin>>name;
    cout<<"Enter age:";
    cin>>age;
}

void employee::put()
{
    cout<<"Name:"<<name<<"\n";
    cout<<"Age:"<<age<<"\n";
}

void main()
{
    employee e[50];
    int n;
    cout<<"Enter the No of Employees:";
    cin>>n;
    cout<<"Enter the Details:";
    for(int i=0;i<n;i++)
```

```

    e[i].get();
    cout<<"\nDetails of"<<n<<"Employees";
    for(i=0;i<n;i++)
    {
        cout<<"\nEmployee"<<i+1<<"\n";
        e[i].put();
    }
}

```

Friendly functions

Private members are accessed only within the class they are declared. Friend function is used to access the private and protected members of different classes. It works as bridge between classes.

- Friend function must be declared with **friend** keyword.
- Friend function must be declare in all the classes from which we need to access private or protected members.
- Friend function will be defined outside the class without specifying the class name.
- Friend function will be invoked like normal function, without any object.

Syntax:

```
friend returntype functionname(Arg list)
```

Program:

```

#include<iostream.h>
class second;
class first
{

```



```
int a;
public:
    void set(int x)
    {
        a=x;
    }
    friend void max(first,second);
};
class second
{
    int b;
    public :
        void set(int y)
        {
            b=y;
        }
        friend void max(first,second);
};
void max(first m,second n)
{
    if(m.a>=n.b)
        cout<<m.a;
    else
        cout<<n.b;
}
void main()
{
    first f;
    f.set(10);
```

```
second s;  
s.set(20);  
max(f,s);  
}
```

Constructors and destructors: Constructors

Constructors:

Constructor is a special function used to initialize class data members or we can say constructor is used to initialize the object of class.

- Constructor name class name must be same.
- Constructor doesn't return value.
- Constructor is invoked automatically, when the object of class is created.

Characteristics

- Should be in the public section
- Invoked automatically
- Do not have any return value
- Cannot be inherited
- Can have arguments
- Cannot be virtual
- Cannot refer to their address
- Make implicit calls to the operators new and delete.

Types of Constructor

- Default Constructor
- Parameterize Constructor
- Copy Constructor

Construct without parameter is called default constructor.

Example of C++ default constructor

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

class Student
{

    int Roll;
    char Name[25];
    float Marks;

    public:

    Student()      //Default Constructor
    {
        Roll = 1;
        strcpy(Name,"Kumar");
        Marks = 78.42;
    }

    void Display()
    {
        cout<<"\n\tRoll : "<<Roll;
        cout<<"\n\tName : "<<Name;
        cout<<"\n\tMarks : "<<Marks;
    }

};

void main()
{
```

Details

```
Student S;      //Creating Object  
  
S.Display();    //Displaying Student
```

```
}
```

Output :

```
Roll : 1  
Name : Kumar  
Marks : 78.42
```

Construct without parameter is called default constructor.

Example of C++ default constructor

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

Constructor

Student() **//Default**

```
{  
    Roll = 1;  
    strcpy(Name,"Kumar");  
    Marks = 78.42;  
}  
  
void Display()  
{  
    cout<<"\n\tRoll : "<<Roll;  
    cout<<"\n\tName : "<<Name;  
    cout<<"\n\tMarks : "<<Marks;  
}
```

```
};
```

```
void main()  
{
```

Student S; **//Creating Object**

S.Display(); **//Displaying**

Student Details

```
}
```

Output :

Roll : 1

Name : Kumar

Marks : 78.42

Parameterized Constructors

- Parameters are arguments to the Constructor
- This can be done in 2 ways
 - By calling the Constructor explicitly
 - Class-name obj=constructorname(arg list);
 - By calling the Constructor implicitly
 - Class-name obj(arg list);

Program:

```
#include<iostream.h>
class assign
{
    int a,b;
public:
    assign(int x,int y)
    {
        a=x;
        b=y;
    }
    void show()
    {
        cout<<"\nNumber1:"<<a;
        cout<<"\nNumber2:"<<b;
    }
}
```

```
};  
void main()  
{  
    assign f(34,35);  
    f.show();  
    assign s=assign(75,76);  
    s.show();  
}
```

Copy Constructor

- A reference variable is used as an argument to copy constructor
- Constructor contains the address value of another object or a variable as its argument.

Program

```
#include<iostream.h>  
class copy  
{  
    int y;  
public:  
    copy()  
    {  
        cout<<"\nNo Arguments";  
    }  
    copy(int i)  
    {  
        y=i;  
    }  
    copy(copy &x)  
    {
```

```
        y=x.y;
    }
    void display()
    {
        cout<<"\nValues:"<<y;
    }
};

void main()
{
    copy a(111);
    copy b(a);
    copy c=a;
    copy d; d=a;
    cout<<"\nValue of a:";
    a.display();
    cout<<"\nValue of b:";
    b.display();
    cout<<"\nValue of c:";
    c.display();
    cout<<"\nValue of d:";
    d.display();
}
```

Multiple constructors in a class

- More than one Constructor with in a class is called Multiple Constructor
- It is also known as Constructor Overloading

Program

Prepared by Dr.S.Hemalatha, Department of Commerce (Computer Application) ,
KAHE


```
#include<iostream.h>
class Method
{
public:
    Method()
    {
        cout<<"\nNo Arguments";
    }
    Method(int i)
    {
        cout<<"\nInteger Argument:"<<i;
    }
    Method(double i)
    {
        cout<<"\nDouble Argument:"<<i;
    }

    Method(char i)
    {
        cout<<"\nCharacter Argument:"<<i;
    }
};

void main()
{
    Method b;
    Method b1(5);
    Method b2(6.5);
    Method b3('c');
```

```
}
```

Dynamic Constructor

Basically, it's a way of constructing an object based on the run-time type of some existing object. It basically uses standard virtual functions/polymorphism.

```
class base
{
public:
virtual base* create() = 0;
virtual base* clone() = 0;
protected:
base();
base(const base&);
};

virtual der1 : public base
{
public:
base* create() { return new der1; }
base* clone() { return new der1(*this); }
};

virtual der2 : public base
{
public:
base* create() { return new der2; }
base* clone() { return new der2(*this); }
};

int main()
{
base* b = new der1;
```

```
base* b1 = b->create();  
base* b2 = b->clone();  
}
```

Destructor

Constructor allocates the memory for an object. Destructor deallocates the memory occupied by an object. Like constructor, destructor name and class name must be same, preceded by a tilde(~) sign. Destructor takes no argument and has no return value.

Constructor is invoked automatically when the object is created. Destructor is invoked when the object goes out of scope. In other words, Destructor is invoked, when the compiler comes out from the function where an object is created.

- Destructor destroys the objects that have been created by a constructor
- It is also a special member function
- Its also same name as the class name preceded by a tilde

Syntax:

```
~class-name();
```

- It never takes any arguments and has no return value
- Automatically invoked by compiler at the end of program
- delete is used for free memory

Program:

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

Prepared by Dr.S.Hemalatha, Department of Commerce (Computer Application) ,
KAHE

```
int y;
public:
    copy()
    {
        count++;
        cout<<"\nNo Objects Created:"<<count;
    }
    ~copy()
    {
        cout<<"\nNo Objects Deleted:"<<count;
        count--;
    }
};

void main()
{
    copy a,b,c,d;
}
```



KARPAGAM ACADEMY OF HIGHER EDUCATION
(Deemed University Established Under Section 3 of UGC Act 1956)
Coimbatore – 21
(For the candidates admitted from 2016 onwards)
DEPARTMENT OF COMMERCE (CA)

SUBJECT : OBJECT ORIENTED PROGRAMMING WITH C++

SEMESTER : III

SUBJECT CODE: 16CCU302

CLASS : II B.COM CA

POSSIBLE QUESTIONS – UNIT III

PART A (1 Mark)

(Online Examinations)

PART B (2 Marks)

1. Define Class.
2. Define Object.
3. How will you Specify a class
4. How will you define member functions
5. What is pointers to Members
6. Define Constructor
7. Define Destructor
8. How many types of Constructor
9. Define Static data members
10. What is Static Member Function
11. What is Nesting of Member Functions
12. What is Jump Students
13. What is Goto Statements

PART C (6 Marks)

1. Explain Member functions in a class
2. Explain Static Data members with example.
3. Explain Static Member functions with example
4. Describe Constructor and Destructor
5. Discuss Private Member Functions
6. Write a program to subtract two numbers (member functions should be define outside the class)
7. Elucidate dynamic destructor with example.
8. Explain member function and nesting of member function
9. What is constructor? Explain the types of constructor with an suitable example
10. Engrave on friend function with suitable example.



KARPAGAM ACADEMY OF HIGHER EDUCATION

(Deemed University Established Under Section 3 of UGC Act 1956)

Coimbatore - 641021.

(For the candidates admitted from 2016 onwards)

DEPARTMENT OF COMMERCE (CA)

SUBJECT : OBJECT ORIENTED PROGRAMMING WITH C++

SEMESTER : III

SUBJECT CODE: 16CCU302

CLASS : II B.COM CA

UNIT IV

Operator Over Loading -Type Conversion – Introduction – Defining Operator Over Loading – Over Loading Unary & Binary Operators – Over Loading Binary Operators using Friends – Manipulation of String Using Operators – Rules for Over Loading Operators – Types – Conversions – Inheritance – Extending Classes – Defining Derived Classes – Single, Multi Level Multiple, Hierarchical & Hybrid Inheritance – Virtual Base Classes – Abstract Classes.

Unit – IV

Operator overloading: Defining operator overloading

- The process of giving special meaning to a method or an operator is called Operator Overloading
- Overloading is the process of adding an extra or additional operation to an existing operation
- Overloading consist of same name but differ in their argument list, Number of argument or both.
- There are two types of overloading
 - Function overloading
 - Operator overloading

Method Overloading

- Change the meaning of a function
- The name of the function is same but differ in their operation differ in their arguments list
- Function overloading is done by using various number arguments to a function
- Function perform different operation based on the requirements

Program:

```
#include<iostream.h>
class over
{
    public:
        void add(int a,int b)
        {
            cout<<"\nAddition of integer:"<<a+b;
        }
        void add(double a,double b)
```



```
{
    cout<<"\nAddition of double:"<<a+b;
}

void add(int a,double b)
{
    cout<<"\nAddition of integer & double:"<<a+b;
}
void add(double a,int b)
{
    cout<<"\nAddition of double and integer:"<<a+b;
}
void add(int a)
{
    cout<<"\nOne Argument:"<<a;
}
};

void main()
{
    over b;
    b.add(5,6);
    b.add(8.2,7.8);
    b.add(7,8.3);
    b.add(8.3,7);
    b.add(111);
}
```

Operator Overloading

- Mechanism of giving special meaning to an operator
- It creates a new definition for most c++ operators

- Semantics of an operator is extended
- It does not change the meaning of the operator

Rules for Overloading Operators:

1. Only existing operators can be overloaded. New operators cannot be created
2. The overloaded operator must have at least one operand that is of user-defined type
3. Can not be able to change the predefined meaning of the Operator.
4. An overloaded operator follows the syntax rules of the original operators. They can not be overridden
5. Some Operators that can not be overloaded.
6. Certain Operators can not be overloaded using the friend Function.

Operators Cannot be Overloaded

- Membership operators (.)
- Pointer-to-member operator (.*)
- Scope resolution operator (::)
- Size of operator (sizeof)
- Conditional operator (?:)

Operators Cannot be Overloaded Using friend Function

- Assignment operator (=)
- Function call operator (())
- Subscripting operator ([])
- Class member access operator (->)

Defining Operator Overloading

- Done with the help of a special function, *operator function*, which describes the task

Syntax:

- **Declaration:**

RT operator operatorsymbol(argument list)

- **Definition:**

RT classname :: operator(op-arglist)

```
{  
    function body  
}
```

Example:

void operator –()

- Operator function must be either member functions or friend function
- Difference: a friend function will have only 1 argument for unary operators and 2 arguments for binary operator

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

Example

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

```
class Add
```

```
{  
    int lat,log;  
    public:  
        Add(){}  
        Add(int l,int lt)  
        {  
            lat=l;
```

```
        log=lt;
    }
    void show()
    {
        cout<<lat<<" ";
        cout<<log<<" ";
    }
    Add operator -(Add o);
};
Add Add::operator -(Add o)
{
    Add t;
    t.lat=o.lat+lat;
    t.log=o.log+log;
    return t;
}

void main()
{
    Add a(10,20),b(30,50);
    a.show();
    b.show();
    a=a-b;
    a.show();
}
```

Overloading unary operators

Overloading Unary Operators:

- The operator has only one Operand.

- Unary operators are unary +, unary -, ++, --, this operator changes the sign of the operand.

Program

```
#include<iostreams.h>
class space
{
    int x;
    int y;
    int z;
public:
    void get(int a,int b,int c);
    void display(void);
    void operator -();
};
void space::get(int a,int b,int c)
{
    x=a;
    y=b;
    z=c;
}
void space::display(void)
{
    cout<<x<<"\n";
    cout<<y<<"\n";
    cout<<z<<"\n";
}
void space::operator -()
{
    x=-x;
```

```
y=-y;
z=-z;
}
void main()
{
    space s;
    s.getdata(10,-20,30);
    cout<<"\nValues before Call Operator Overloading\n";
    s.display();
    -s;
    cout<<"\nValues After Call Operator Overloading\n";
    s.display();
}
```

Overloading binary operators

- The operator has two Operand.

Program:

```
#include<iostreams.h>
class Time
{
    int h,m;
    public:
        Time(){}

        Time(int hr,int min)
        {
            h=hr;
            m=min;
        }

        void display(void);
}
```

```
    Time operator+(Time);
};
void Time::display(void)
{
    cout<<h<<"hours and"<<m<<" Min\n";
}
Time Time::operator+(Time t)
{
    Time t1;
    t1.m=m+t.m;
    int bal=t1.m/60;
    t1.m=t1.m%60;
    t1.h=h+t.h+bal;
    return(t1);
}
void main()
{
    Time h1,h2,h3;
    h1=Time(2,50);
    h2=Time(2,50);
    h3=h1+h2;
    cout<<"\nTime t1:";
    h1.display();
    cout<<"\nTime t2:";
    h2.display();
    cout<<"\nTime t3:";
    h3.display();
}
```

Overloading binary operators using friends

- Non member function of a class can be able to access the private members of a class through friend function
- Friend Function are created with the keyword friend
- A friend function requires two arguments to be explicitly passed to it.

Program:

```
#include <iostream.h>
#include <conio.h>
class Point
{
    int x, y;
public:
    Point()
    {}
    Point(int px, int py)
    {
        x = px;
        y = py;
    }
    void show()
    {
        cout << x << " ";
        cout << y << "\n";
    }
    friend Point operator+(Point op1, Point op2); // now a friend
    Point operator=(Point op2);
};
```



```
// Now, + is overloaded using friend function.
Point operator+(Point op1, Point op2)
{
    Point temp;
    temp.x = op1.x + op2.x;
    temp.y = op1.y + op2.y;
    return temp;
}
// Overload assignment for Point.
Point Point::operator=(Point op2)
{
    x = op2.x;
    y = op2.y;
    return op2; // i.e., return object that generated call
}
int main()
{
    clrscr();
    Point ob1(10, 20), ob2( 5, 30);
    ob1 = ob1 + ob2;
    ob1.show();
    return 0;
}
```

type conversions

- Type Conversion is the process of change the data type of variable.
- When constants and variables of different types are mixed in an expression, C applies automatic type conversion to the operand as per certain rules.

➤ In the Assignment Operation the type of data to the right of an assignment operator is automatically converted to the type of the variable on the left.

Example: `int m;
float x=5.89;
m=x;`

The value of m is 5 since the fraction part is truncated.

➤ The Compiler does not support automatic type conversion for user defined data types. Since the type conversion should be performed explicitly.

➤ Three types of situations arises in the data conversion between incompatible types:

- ❖ Conversion from basic type to class type
- ❖ Conversion from class type to basic type
- ❖ Conversion from one class type to another class type.

Conversion from basic type to class type

➤ In this left hand operand of = operator is always a class object.

Example:

```
String s1,s2;  
char *name1="C++ Programming";  
char *name2="C Programming";  
s1=String(name1);  
s2=name2;
```

String is the class, name1 is char variable which is converted explicitly in the statement

```
s1=String(name1);
```

name2 is char variable which is converted implicitly by call the constructor in the statement

```
s2=name2;
```

The constructor used for type conversion takes a single argument whose type is to be converted. This conversion is done by overloaded = operator.

Conversion from class type to basic type

- In this left hand operand of = operator is always a variable type or basic type.
- The constructor performs a fine job in conversion from basic to class type.
- The conversion of class to basic model is done by overloaded casting operator.
- The general form of an overloaded casting operator function is

```
operator typename()  
{  
    .....  
    .....//function statement  
}
```

Example:

```
String :: operator double()  
{  
    double d=0;  
    d=s[0]+s[1];  
    return(d);  
}
```

String is a class converted to basic type double, where s is a String class object

```
String s1,s2;  
float c1,c2;  
c1=float(s1);  
c2=s2;
```

c1 is float variable , String is the class which is converted explicitly in the statement

```
c1=float(s1);
```

c2 is float variable , String is the class which is converted implicitly in the statement

```
c2=s2;
```

The casting operator function should satisfy the following conditions:

- It must be a class member
- It must not specify a return type
- It must not have any arguments.

Conversion from one class type to another class type.

Example:

```
S1= S2 // objects of different types
```

- S1 and S2 are the object of two different classes class X and class Y.
- The class Y type is converted into X type.
- Y is known as the source class and X is known as the designation class.
- This type of conversion is performed using either a contractor or a conversion function
- Casting function is of the form

```
operator typename()
```

Type Conversion Table

Conversions Required	Conversion takes place in	
	Source class	Designation class
Basic →class	Not applicable	Constructor
Class→Basic	Casting operator	Not applicable
Class→class	Casting operator	Constructor

Inheritance :- Inheritance

- Sharing the properties of one class by the other.
- Ability of a new class to be created, from an existing class by extending it, is called inheritance.
- Different kinds of objects often have a certain amount in common with each other.
- Object-oriented programming allows classes to inherit commonly used state and behavior from other classes.
- A class which provides the data is called Base class
- A class receives the data is called Derived class
- No changes are made to the base class

Advantage of Inheritance:

- Reusability of code
- Save a lot of time and efforts, retyping the same
- Data and methods of super class are physically available to its subclasses

Forms of Inheritance

In C++ there are 5 forms of inheritance.

- Single Inheritance
- Multiple Inheritance
- Multilevel Inheritance
- Hierarchical Inheritance
- Hybrid Inheritance

Defining derived classes

- Derived class can be defined by specifying the relationship with the base class in addition to its own details.
- : (colon) operator is used for inheritance.

Syntax:

```
class derived-class-name : visibility-mode base-class-name
{
    .....
    ..... //derived class member functions
    .....
};
```

- The colon indicates that the derived-class-name is derived from the base-class-name.
- The visibility-mode is optional, if presents private or public or protected access specifies can be specified
- By default visibility-mode is private.
- The visibility-mode specifies whether the features of the base class are privately derived or publicly derived.

Example:

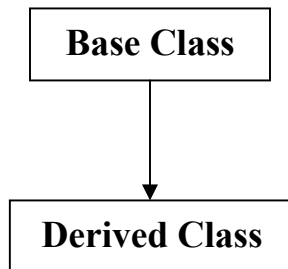
```
class ABC
{
    .....
    .....// base class members
}
class der : private ABC //Privately inherited from class ABC
{
    .....
    .....// derived class members
}
```

```
class der : public ABC //Publicly inherited from class ABC
{
    .....
    .....// derived class members
}

class der : ABC //Privately inherited from class ABC by default
{
    .....
    .....// derived class members
}
```

Single, multilevel, multiple, hierarchical inheritance

➤ Single inheritance consist of single base class and single derived class



Syntax:

```
class derived-class-name : visibility-mode base-class-name
{
    .....
}
```

```
.....// derived class members  
}
```

The colon (:), indicates that the class derived-class-name is derived from the class base-class-name.

Program:

```
#include<iostream.h>  
class Rectangle  
{  
    private:  
        float length ; // This can't be inherited  
    public:  
        float breadth ; // The data and member functions are inheritable  
        void Enter_lb(void)  
        {  
            cout << "\n Enter the length of the rectangle : ";  
            cin >> length ;  
            cout << "\n Enter the breadth of the rectangle : ";  
            cin >> breadth ;  
        }  
        float Enter_l(void)  
        {  
            return length ;  
        }  
}; // End of the class definition  
class Rectangle1 : public Rectangle  
{  
    private:  
        float area ;
```



```

public:
    void Rec_area(void)
    {
        area = Enter_l( ) * breadth ;
    }
    void Display(void)
    {
        cout << "\n Length = " << Enter_l( ) ;
        cout << "\n Breadth = " << breadth ;
        cout << "\n Area = " << area ;
    }
};

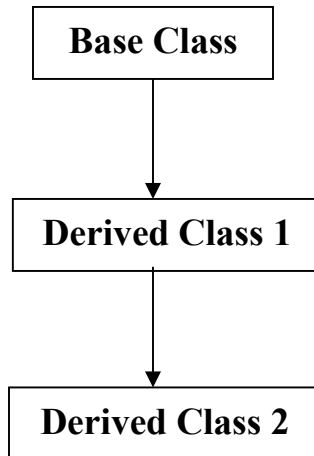
void main(void)
{
    Rectangle1 r1 ;
    r1.Enter_lb( );
    r1.Rec_area( );
    r1.Display( );
}
    
```

Visibility of Inherited Members

Base visibility	Derived class visibility		
	public derivation	private derivation	protected derivation
private	Not Inherited	Not Inherited	Not Inherited
protected	protected	private	protected
public	public	private	protected

Multilevel Inheritance:

➤ C++ also provides the facility of multilevel inheritance, according to which the derived class can also be derived by another class, which in turn can further be inherited by another and so on.



Syntax:

```
class derived-class-name1 : visibility-mode base-class-name
{
    .....
    .....// derived class members
}
class derived-class-name2 : visibility-mode derived-class-name1
{
    .....
    .....// derived class members
}
```

derived-class-name1 is inherited from base-class-name then the derived-class-name2 is inherited from derived-class-name1.

Program:

```
#include<iostream.h>
class Base
{
    protected:
        int b;
    public:
        void EnterData( )
        {
            cout << "\n Enter the value of b: ";
            cin >> b;
        }
        void DisplayData( )
        {
            cout << "\n b = " << b;
        }
};
class Derive1 : public Base
{
    protected:
        int d1;
    public:
        void EnterData( )
        {
            Base:: EnterData( );
            cout << "\n Enter the value of d1: ";
            cin >> d1;
```

```
    }
    void DisplayData( )
    {
        Base::DisplayData();
        cout << "\n d1 = " << d1;
    }
};

class Derive2 : public Derive1
{
    private:
        int d2;
    public:
        void EnterData( )
        {
            Derive1::EnterData( );
            cout << "\n Enter the value of d2: "; cin >> d2;
        }
        void DisplayData( )
        {
            Derive1::DisplayData( );

            cout << "\n d2 = " << d2;
        }
};

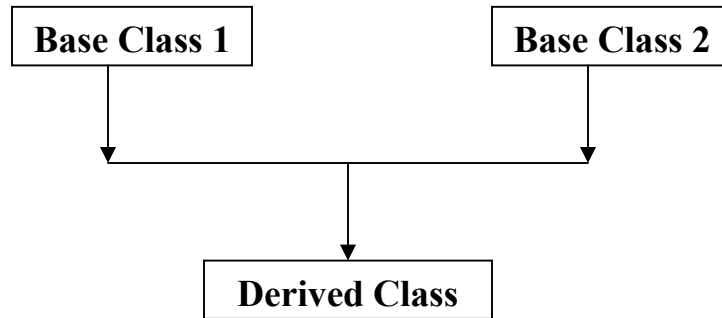
int main( )
{
    Derive2 objd2;
    objd2.EnterData( );
    objd2.DisplayData( );
}
```

```
    return 0;  
}
```

Multiple

Inheritance

When a class is inherited from more than one base class, it is known as multiple inheritance.



Syntax:

```
class derived-class-name : visibility-mode base-class-name1, visibility-  
mode base-class-name2  
{  
    .....  
    .....// derived class members  
}
```

derived-class-name is derived from two base classes namely base-class-name1 and base-class-name1

Program:

```
#include<iostream.h>  
class Circle // First base class  
{  
    protected:  
        float radius ;  
    public:
```

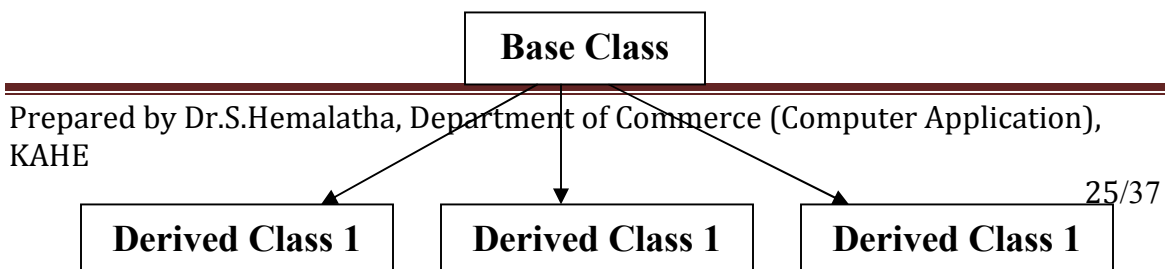
```
void Enter_r(void)
{
    cout << "\n\t Enter the radius: "; cin >> radius ;
}
void Display_ca(void)
{
    cout << "\t The area = " << (22/7 * radius*radius) ;
}
};
class Rectangle // Second base class
{
    protected:
    float length, breadth ;
    public:
        void Enter_lb(void)
        {
            cout << "\t Enter the length : ";
            cin >> length ;
            cout << "\t Enter the breadth : " ;
            cin >> breadth ;
        }
        void Display_ar(void)
        {
            cout << "\t The area = " << (length * breadth);
        }
};
class Cylinder : public Circle, public Rectangle // Derived class,
inherited
{ // from classes Circle & Rectangle
```

```
public:
    void volume_cy(void)
    {
        cout << "\t The volume of the cylinder is: "<< (22/7*
radius*radius*length) ;
    }
};

void main(void)
{
    Circle c ;
    cout << "\n Getting the radius of the circle\n" ;
    c.Enter_r( );
    c.Display_ca( );
    Rectangle r ;
    cout << "\n\n Getting the length and breadth of the rectangle\n\n";
    r.Enter_lb( );
    r.Display_ar( );
    Cylinder cy ;
    cout << "\n\n Getting the height and radius of the cylinder\n";
    cy.Enter_r( );
    cy.Enter_lb( );
    cy.volume_cy( );
}
```

Hierarchical Inheritance:

- When two or more classes are derived from a single base class, then Inheritance is called the hierarchical inheritance.
- In this type there exists a hierarchical relation in the inheritance.



Syntax:

```
class derived-class-name1 : visibility-mode base-class-name
{
    .....
    .....// derived class members
}
class derived-class-name2 : visibility-mode base-class-name
{
    .....
    .....// derived class members
}
```

derived-class-name1, derived-class-name2 are two derived class derived from the class base-class-name.

Example:

```
#include<iostream.h>
const int len = 20 ;
class student
{
    private:
        char F_name[len] , L_name[len] ;
        int age,roll_no ;
    public:
        void Enter_data(void)
        {
            cout << "\n\t Enter the first name: " ; cin >> F_name ;
            cout << "\t Enter the second name: "; cin >> L_name ;
        }
}
```



```
    cout << "\t Enter the age: " ; cin >> age ;
    cout << "\t Enter the roll_no: " ; cin >> roll_no ;
}
void Display_data(void)
{
    cout << "\n\t First Name = " << F_name ;
    cout << "\n\t Last Name = " << L_name ;
    cout << "\n\t Age = " << age ;
    cout << "\n\t Roll Number = " << roll_no ;
}
};
class arts : public student
{
    private:
        char asub1[len] ;
        char asub2[len] ;
        char asub3[len] ;
    public:
        void Enter_data(void)
        {
            student :: Enter_data( );
            cout << "\t Enter the subject1 of the arts student: ";
            cin >> asub1 ;
            cout << "\t Enter the subject2 of the arts student: ";
            cin >> asub2 ;
            cout << "\t Enter the subject3 of the arts student: ";
            cin >> asub3 ;
        }
        void Display_data(void)
```

```
{
    student :: Display_data( );
    cout << "\n\t Subject1 of the arts student = " << asub1 ;
    cout << "\n\t Subject2 of the arts student = " << asub2 ;
    cout << "\n\t Subject3 of the arts student = " << asub3 ;
}
};
class science : public student
{
    private:
        char csub1[len], csub2[len], csub3[len] ;
    public:
        void Enter_data(void)
        {
            student :: Enter_data( );
            cout << "\t Enter the subject1 of the science student: ";
            cin >> csub1;
            cout << "\t Enter the subject2 of the science student: ";
            cin >> csub2 ;
            cout << "\t Enter the subject3 of the science student: ";
            cin >> csub3 ;
        }
        void Display_data(void)
        {
            student :: Display_data( );
            cout << "\n\t Subject1 of the science student = " << csub1 ;
            cout << "\n\t Subject2 of the science student = " << csub2 ;
            cout << "\n\t Subject3 of the science student = " << csub3 ;
        }
}
```

```
};  
void main(void)  
{  
    arts a ;  
    cout << "\n Entering details of the arts student\n" ;  
    a.Enter_data( );  
    cout << "\n Displaying the details of the arts student\n" ;  
    a.Display_data( );  
    science s ;  
    cout << "\n\n Entering details of the science student\n" ;  
    s.Enter_data( );  
    cout << "\n Displaying the details of the science student\n" ;  
    s.Display_data( );  
}
```

Hybrid inheritance

➤ Combination of multiple and multilevel inheritance is called hybrid inheritance.

Syntax:

```
class derived-class-name1 : visibility-mode base-class-name  
{  
    .....  
    .....// derived class members  
}  
class derived-class-name2 : visibility-mode base-class-name  
{  
    .....  
    .....// derived class members
```

```
}  
class derived-class-name3 : visibility-mode derived-class-name1,  
visibility-mode derived-class-name2  
{  
    .....  
    .....// derived class members  
}
```

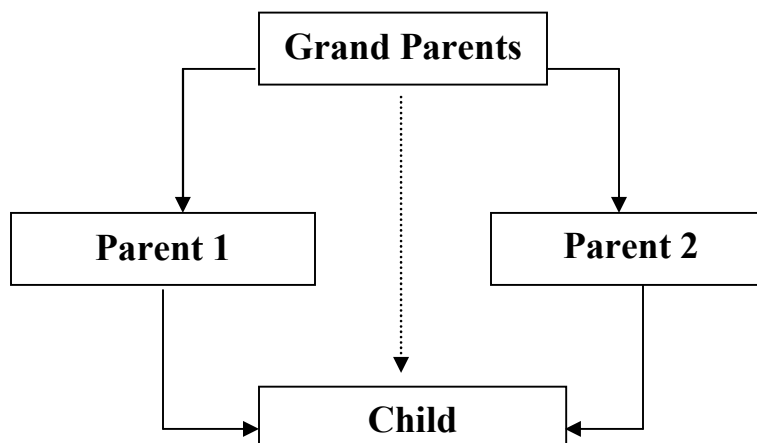
Example:

```
#include<iostream.h>  
#include<conio.h>  
class stu  
{  
    protected:  
        int rno;  
    public:  
        void get_no(int a)  
        {  
            rno=a;  
        }  
        void put_no(void)  
        {  
            cout<<"Roll no"<<rno<<"\n";  
        }  
};  
class test:public stu  
{  
    protected:  
        float part1,part2;  
    public:
```

```
void get_mark(float x,float y)
{
    part1=x;
    part2=y;
}
void put_marks()
{
    cout<<"Marks
obtained:"<<"part1="<<part1<<"\n"<<"part2="<<part2<<"\n";
}
};
class sports
{
    protected:
        float score;
    public:
        void getscore(float s)
        {
            score=s;
        }
        void putscore(void)
        {
            cout<<"sports:"<<score<<"\n";
        }
};
class result: public test, public sports
{
    float total;
    public:
```

```
void display(void);  
};  
void result::display(void)  
{  
    total=part1+part2+score;  
    put_no();  
    put_marks();  
    putscore();  
    cout<<"Total Score="<<total<<"\n";  
}  
int main()  
{  
    clrscr();  
    result stu;  
    stu.get_no(111);  
    stu.get_mark(27.5,33.0);  
    stu.getscore(10.0);  
    stu.display();  
    return 0;  
}
```

Virtual base classes



- In some situations which requires the use of both multiple and multilevel inheritance
- Consider a situation where all three kinds of inheritance, namely multiple, multilevel and hierarchical inheritance are involved.
- In the above figure 'Child' has two base classes 'Parent1' and 'Parent2' which themselves have common base class 'Grand Parents'.
- The 'Child' inherits the traits of 'Grand Parent' via two separate paths.
- It can also inherit directly as shown by broken line.
- The 'Grand Parents' is sometimes referred as indirect base class.
- In the above case there exist a problem all the public and protected members of 'Grand Parents' are inherited into 'Child' twice, first via 'Parent 1' and again via 'Parent 2'. This introduces ambiguity and should be avoided.
- The duplication of inherited members due to these multiple paths can be avoided by making the common base class as virtual base class while declaring the direct or intermediate base class.

Syntax:

```
class base-class-name
{
    .....
    .....// base class members Grand Parents
}

class derived-class-name1 : virtual visibility-mode base-class-name
{
    .....
```

```
.....// derived class members Parent1
}
class derived-class-name2 : visibility-mode virtual base-class-name
{
    .....
    .....// derived class members Parent2
}
class derived-class-name3 : visibility-mode derived-class-name1,
visibility-mode derived-class-name2
{
    .....
    .....// derived class members Child
}
```

➤ When a class is made 'virtual' base class, c++ takes necessary care to see that only one copy of that class is inherited, regardless of how many inheritance paths exist between the virtual base class and a derived class.

Program:

```
#include<iostream.h>
#include<conio.h>
class stu
{
    protected:
        int rno;
    public:
        void get_no(int a)
        {
            rno=a;
        }
}
```



```
void put_no(void)
{
    cout<<"Roll no"<<rno<<"\n";
}
};

class test:virtual public stu//Virtually inherited
{
    protected:
        float part1,part2;
    public:
        void get_mark(float x,float y)
        {
            part1=x;
            part2=y;
        }
        void put_marks()
        {
            cout<<"Marks
obtained:\npart1="<<part1<<"\n"<<"part2="<<part2<<"\n";
        }
};

class sports: public virtual stu
{
    protected:
        float score;
    public:
        void getscore(float s)
        {
            score=s;
        }
};
```

```
    }  
    void putscore(void)  
    {  
        cout<<"sports:"<<score<<"\n";  
    }  
};  
class result: public test, public sports  
{  
    float total;  
    public:  
        void display(void);  
};  
void result::display(void)  
{  
    total=part1+part2+score;  
    put_no();  
    put_marks();  
    putscore();  
    cout<<"Total Score="<<total<<"\n";  
}  
int main()  
{  
    clrscr();  
    result stu;  
    stu.get_no(123);  
    stu.get_mark(27.5,33.0);  
    stu.getscore(6.0);  
    stu.display();  
    return 0;  
}
```

}

Abstract classes

- abstract keyword is used to create abstract class.
- An abstract class is one that is not used to create object
- An abstract class is designed only to act as a base class.



KARPAGAM ACADEMY OF HIGHER EDUCATION

(Deemed University Established Under Section 3 of UGC Act 1956)

Coimbatore - 641021.

(For the candidates admitted from 2016 onwards)

DEPARTMENT OF COMMERCE (CA)

SUBJECT : OBJECT ORIENTED PROGRAMMING WITH C++

SEMESTER : III

SUBJECT CODE: 16CCU302

CLASS : II B.COM CA

UNIT V

Pointers, Virtual Functions & Polymorphism – Pointers to Object - Pointers to Derived Classes – Virtual Functions .Working with Files – Classes for File Stream Operations – Opening and Closing a File – File Pointers & their Manipulations - Sequential I/O Operations.

Unit – V

Pointers: Pointers to objects

- Object pointers are useful in creating objects at run time.
- Pointer objects are useful to access the public members of an object.

Syntax:

```
classname *ptr-variable;
```

Example:

```
#include<iostream.h>
class item
{
    int code;
    float price;
public:
    void getdata(int a, float b)
    {
        code=a;
        price=b;
    }
    void show()
    {
        cout<<"Code:"<<code;
        cout<<"\nPrice:"<<price;
    }
};
void main()
{
    item *p=new item[3];
```

```

item *d=p;
int x,i;
float y;
for(i=0;i<3;i++)
{
    cout<<"Input Code and Price for Item"<<i+1;
    cin>>x>>y;
    p->getdata(x,y);
    p++;
}
for(i=0;i<3;i++)
{
    cout<<"Item"<<i+1<<endl;
    d->show();
    d++;
}
}

```

this pointer

- this is used to represent current object.
- this is a keyword.
- this is an object that invokes a member function.
- This unique pointer is automatically passed to a member function when it is called.
- The pointer this acts as an implicit argument to all the member functions.

Syntax:

this - > variable or function name;

Example:

```

    this->a;
    return this;

```

Program:

```

#include<iostream.h>
class person
{
    char name[20];
    float age;
public:
    person(char *s, float a)
    {
        strcpy(name,s);
        age=a;
    }
    person &person :: greater(person &x)
    {
        if(x.age>=age)
            return x;
        else
            return *this;
    }
    void display()
    {
        cout<<"Name : "<<name;
        cout<<"\nAge : "<<age;
    }
};

void main()

```

```
{
    person p1("Raja",42.78),p2("Ram",35.8),p3("Arun",45.3);
    person p=p1.greater(p3);
    cout<<"Elder Person is :\n ";
    p.display();
    p=p1.greater(p2);
    cout<<"Elder Person is :\n ";
    p.display();
}
```

Pointers to derived classes

- Pointers are used not only to base class but also to derived class.
- Pointers to objects of a base class are type-compatible with pointers to objects of derived class.
- A single pointer variable can be made to point to object belonging to different classes.

Example:

```
B *ptr;
B b;
D d;
ptr=&b;
```

Program:

```
#include<iostream.h>
class BC
{
    public :
        int b;
        void show()
        {
```



```
        cout<<"b = "<<b<<endl;
    }
};
class DC: public BC
{
    public:
        int d;
        void show()
        {
            cout<<"b = "<<b<<endl;
            cout<<"d = "<<d<<endl;
        }
};
void main()
{
    BC *ptr;
    BC base;
    cout<<"Base Class Pointer Call"<<endl;
    ptr=&base;
    ptr->b=100;
    ptr->show();
    cout<<"Base Class Pointer Call using Derived Class"<<endl;
    DC dc;
    ptr=&dc;
    ptr->b=200;
    ptr->show();
    cout<<"Derived Class Pointer Call"<<endl;
    DC *dptr;
    dptr=&dc;
```

```

dptr->d=100;
dptr->show();
cout<<"Using (DC*)bptr)\n";
((DC *)ptr)->d=200;
((DC *)ptr)->show();
}

```

Virtual functions

- The appropriate member function could be selected while the program is running. This is known as runtime Polymorphism
- Runtime Polymorphism is achieving through virtual function.
- Virtual function is created using the keyword virtual.
- When the function made virtual, C++ determines which function to use at run time based on the type of the object pointed to by the base pointer, rather than the type of the pointer.
- By making the base class pointer to point to different objects, can execute different versions of the virtual function.

Syntax:

```

virtual RT function-name(arg list)
{
    .....
    .....
}

```

Example:

```

virtual void show()
{
    .....
    .....
}

```

Program:

```
#include<iostream.h>
class Base
{
    public :
        void display()
        {
            cout<<"\nDisplay Base Class";
        }
        virtual void show()
        {
            cout<<"Base Class Show";
        }
};
class Derived: public Base
{
    public :
        void display()
        {
            cout<<"\nDisplay Derived Class";
        }
        virtual void show()
        {
            cout<<"Derived Class Show";
        }
};
void main()
{
    Base *ptr;
```

```
Base b;  
Derived d;  
cout<<"\nPointer point to Base Class";  
ptr=&b;  
ptr->display();  
ptr->show();  
cout<<"\nPointer point to Derived Class";  
ptr=&d;  
ptr->display();  
ptr->show();  
}
```

Rules:

1. The virtual function must be members of some class.
2. They can not be static members.
3. They are accessed by using object pointers.
4. A virtual function can be a friend of another class.
5. A virtual function in a base class must be defined, even through it may not be used.
6. The prototype of the base class version of a virtual function and all the derived class versions must be identical. If two functions with the same name have different prototypes, C++ considers them as overloaded function.
7. Can have virtual constructors not have virtual destructors.
8. While a base class pointer can point to any type of derived object, the reverse is not true.
9. When a base class pointer points to a derived class, incrementing or decrementing it will not make it to point to the next object of the derived class

10. If a virtual function is defined in the base class, it need not be necessarily redefined in the derived class.

Pure Virtual Function:

- A pure virtual function is a function declared in a base class that has no definition relative to the base class.
- In this case the compiler requires the derived class to either define the function or redeclare it as pure virtual function.

Syntax:

virtual RT function-name()=0;

Example:

virtual void show()=0;

Managing Console I/O Operations:

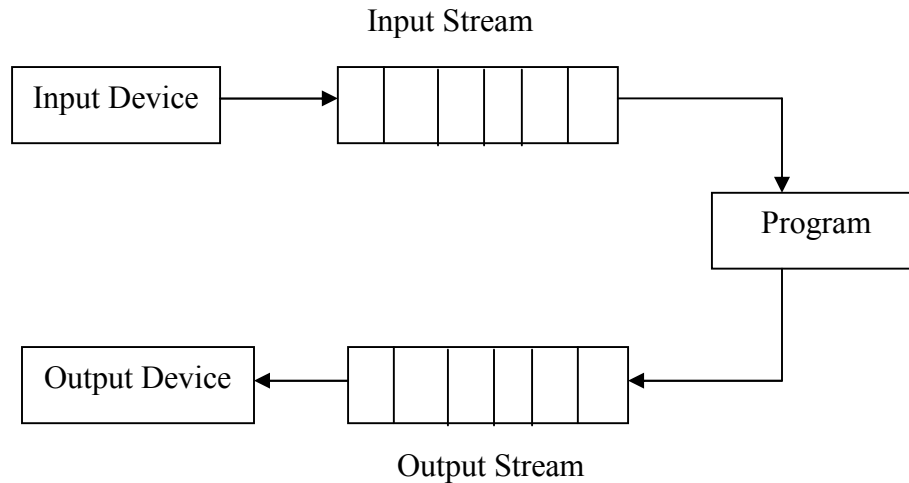
- C++ provides rich set of I/O functions and operations to manage console I/O operations.
- C++ uses the concept stream and stream classes to implement its I/O operations with the console and disk files.

Managing console I/O operations :- C++ streams

C++ Streams:

- The I/O system in C++ is designed to work with a wide variety of devices including terminals, disks, and tape drives.
- The I/O stream supplies an interface to the programmer that independent of the actual device being accessed. This interface is known as stream.
- A stream is a sequence of bytes.
- It acts either as a source from which the input data can be obtained or as a designation to which data can be sent.

- The source stream that provides data to the program is called the input stream and the destination stream that receives output from the program is called the output stream.

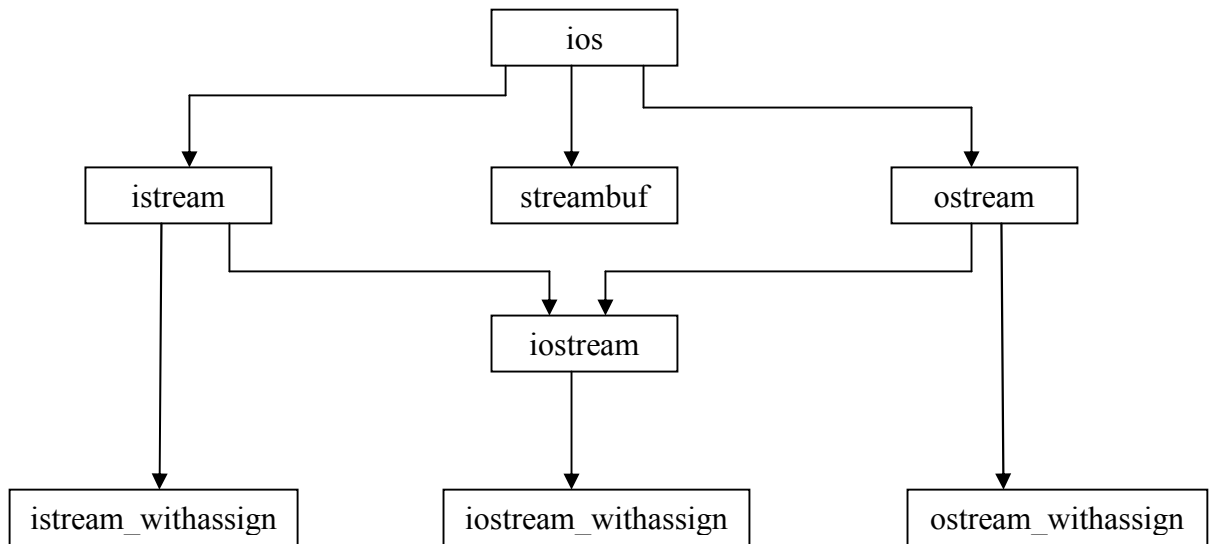


- The data in the input stream can come from the keyboard or any other storage devices.
- The data in the output stream can go to the screen or any other storage devices.
- A stream is an interface between the program and the I/O devices.
- C++ contains pre-defined streams that are automatically opened when a program begins its execution.
- cin and cout also belongs to such streams.
- cin represents the input stream connected to the standard input device.
- cout represents the output stream connected to the standard output device.

C++ stream classes

- The C++ I/O system contains a hierarchy of classes that are used to define various streams to deal with console and disk files. These classes are called stream classes.

- The header file `iostream` should be included in all the programs that communicate with console unit.



- `ios` is the base class for `istream` and `ostream`.
- `istream` and `ostream` are the base class of `iostream`.
- `ios` is declared as virtual so only one copy of its members are inherited by the `iostream`
- `ios` provides the basic support for all I/O operations.
- The class `istream` provides the facilities for formatted and unformatted Input.
- The class `ostream` provides the facilities for formatted output.

Unformatted I/O operations

Overloaded operators `>>` and `<<`

- The objects cin and cout for the input and output of data of various types.
- This is done by overloading the operator >> and <<.
- The operator >> is overloaded in the istream class and << overloaded in the ostream.

Example:

```
cin>>a;  
cout<<a;
```

put() and get() Functions:

- The classes istream and ostream defines two member functions get() and put() to handle the single character input/output operations.
- There are two type of get() function
 - get(char *) – assign the input to a variable
 - get(void) – returns the input character.

Example:

```
cin.get(c); //read and assign the value for c  
c=cin.get(); // returns the read character to c
```

- put() is a member of ostream class, can be used to output a line of text, character by character.

Syntax:

```
cin.put(char *);
```

Example:

```
cout.put('m');
```

Program:

```
#include<iostream.h>  
void main()  
{  
    char c;  
    int ct=0;
```



```

cout<<"Input Text:\n";
cin.get(c);
cout.put(c);
while(c!='\n')
{
    cout.put(c);
    c=cin.get();
    ct++;
}
cout<<"Number of Characters Entered : "<<ct;
}

```

getline() and write() Function:

- getline() and write() is used to read the text line by line.
- The getline() function reads a whole line of text that ends with a newline character.

Syntax:

```

    cin.getline(line,size)
➤ line- variable name
➤ The reading is terminated as soon as either the newline character
'\\n' is encountered or size-1 characters are read.

```

Example:

```

char name[20];
cin.getline(name,20);

```

Program:

```

#include<iostream.h>
void main()
{
    char name[25];
    cout<<"\\nEnter the name:\\n";
}

```

```
cin.getline(name,20);
cout<<"\nName : "<<name;
}
```

- The write() function display an entire line

Syntax:

```
cout.write(line,size)
```

- line- variable name
- The writing is terminated as soon as either the newline character ‘\n’ is encountered or size-1 characters are written.

Example:

```
char name[20]="welcome";
cout.write(name,20);
```

Program:

```
#include<iostream.h>
#include<string.h>
void main()
{
    char name[25];
    int i;
    cout<<"\nEnter the name:\n";
    cin.getline(name,20);
    int l1=strlen(name);
    for(i=1;i<l1;i++)
    {
        cout.write(name,i);
        cout<<endl;
    }
    for(i=l1;i>0;i--)
    {
```

```

    cout.write(name,i);
    cout<<endl;
}
}

```

Formatted console I/O operations

- C++ support a number of features that could be used for formatting the output
- It includes
 - ios class function and flags.
 - Manipulators.
 - User defined output functions.

ios Class Function and Flags:

- The ios class contain a large number of member functions that would help us to format the output in a number of ways.

The members are

Function	Task
width()	To specify the required fields size for displaying an output value
precision()	To specify the number of digits to be displayed after the decimal point of a float values.
fill()	To specify a character that is used to fill the unused portion of a field
setf()	To specify format flags that can control the form of output display.
unsetf()	To clear the flags specified

width():

- To specify the required fields size for displaying an output value

Syntax:

```
cout.width(size)
```

Example:

```
cout.width(5);
```

Program:

```
#include<iostream.h>
void main()
{
    int item[4]={7,12,80,89};
    int cost[4]={75,100,125,90};
    cout.width(5);
    cout<<"Items";
    cout.width(8);
    cout<<"Cost";
    cout.width(20);
    cout<<"Total Values\n";
    int sum=0;
    for(int i=0;i<4;i++)
    {
        cout.width(5);
        cout<<item[i];
        cout.width(8);
        cout<<cost[i];
        cout.width(15);
        cout<<cost[i]*item[i]<<endl;
        sum=sum+cost[i]*item[i];
    }
    cout<<"\nGrand Total=";
```

```
cout.width(2);
cout<<sum<<"\n";
}
```

Output:

Items	Cost	Total Values
7	75	525
12	100	1200
80	125	10000
89	90	8010

Grand Total=19735

precision():

➤ To specify the number of digits to be displayed after the decimal point of a float values.

Syntax:

```
cout.precision(size)
```

Example:

```
cout.precision(2);
```

Program:

```
#include<iostream.h>
#include<math.h>
void main()
{
    cout.precision(3);
    cout.width(7);
    cout<<"Values";
    cout.width(15);
    cout<<"Sqrt values\n";
```

```
for(int i=1;i<5;i++)
{
    cout.width(5);    cout<<i;    cout.width(13);
    cout<<sqrt(i)<<endl;
}
}
```

Output:

Values Sqrt values

1	1
2	1.414
3	1.732
4	2

fill():

➤ To specify a character that is used to fill the unused portion of a field

Syntax:

```
cout.fill(character)
```

Example:

```
cout.fill("*");
```

Program:

```
#include<iostream.h>
void main()
{
    int a=5679;    cout.width(10);
    cout.fill("*");    cout<<"Values";    cout<<a<<"\n";
}
```

Output:

```
****Values5679
```

setf():

- To specify format flags that can control the form of output display.

Syntax:

```
cout.setf(arg1,arg2);
```

- arg1-formatting flags defined in the class ios.
- arg2-formatting flags defined in the class ios. It is also known as bit fields

Flags and Bit Fields

Format required	Flag (arg1)	Bit-field(arg2)
Left-justified O/P	ios::left	ios::adjustfield
Right- justified O/P	ios::right	ios::adjustfield
Padding after sign or base Indicator(+##20)	ios::internal	ios::adjustfield
Scientific notation	ios::scientific	ios::floatfield
Fixed point notation	ios::fixed	ios::floatfield
Decimal base	ios::dec	ios::basefield
Octal base	ios::oct	ios::basefield
Hexadecimal base	ios::hex	ios::basefield

Flags Without Bit Fields

Flag	Meaning
ios::showbase	Use base indicator on output
ios::showpos	Print + before positive numbers
ios::showpoint	Show trailing decimal point and zeroes
ios::uppercase	Use uppercase letters for hex output
ios::skipus	Skip white space on input
ios::unitbuf	Flush all streams after insertion
ios::stdio	Flush stdout and stderr after

Example:

```
cout.setf(ios::left, ios::adjustfield);
```

Program:

```
#include<iostream.h>
#include<math.h>
void main()
{
    cout.fill('*');
    cout.setf(ios::left,ios::adjustfield);
    cout.width(10);  cout<<"Values";
    cout.setf(ios::right,ios::adjustfield);
    cout.width(15);
    cout<<"Sqrt Of Value \n";
    cout.fill('.');  cout.precision(4);
    cout.setf(ios::showpoint);  cout.setf(ios::showpos);
    cout.setf(ios::fixed,ios::floatfield);
    for(int n=1;n<10;n++)
    {
        cout.setf(ios::internal,ios::adjustfield);
        cout.width(5); cout<<n;
        cout.setf(ios::right,ios::adjustfield);
        cout.width(20);
        cout<<sqrt(n)<<endl;
    }
    cout.setf(ios::scientific,ios::floatfield);
    cout<<"sqrt(100)= "<<sqrt(100)<<"\n";
}
```


Output:

Values****Sqrt Of Value

```
+...1.....+1.0000
+...2.....+1.4142
+...3.....+1.7321
+...4.....+2.0000
+...5.....+2.2361
+...6.....+2.4495
+...7.....+2.6458
+...8.....+2.8284
+...9.....+3.0000
sqrt(100)= +1.0000e+01
```

Managing output with manipulators

- The header file `iomanip` provides a set of functions called manipulators which can be used to manipulate the output formats.
- Provide the same features as that of the `ios` member functions and flags.
- The various manipulators are

Manipulator	Meaning	Equivalent
<code>setw(int w)</code>	Set the field width to w	<code>width()</code>
<code>setprecision(int d)</code>	Set the floating point precision to d	<code>precision()</code>
<code>setfill(char c)</code>	Set the fill character to c	<code>fill()</code>
<code>setiosflags(long f)</code>	Set the format flag f	<code>setf()</code>
<code>resetiosflags(long f)</code>	Clear the flag specified by f	<code>unsetf()</code>
<code>endl</code>	Insert new line and flush stream	<code>"\n"</code>

Program:

```
#include<iostream.h>
#include<iomanip.h>
void main()
{
    cout.setf(ios::showpoint);
    cout<<setw(5)<<"n"<<setw(15)<<"Inverse  of  n"<<setw(15)<<"Sum  of
terms"<<endl<<endl;
    double term, sum;
    for(int n=1;n<10;n++)
    {
        term=1.0/float(n);
        sum=sum+term;
    }
    cout<<setw(5)<<n<<setw(14)<<setprecision(2)<<setiosflags(ios::scientific)<
<term<<setw(13)<<resetiosflags(ios::scientific)<<sum<<endl;
    }
}
```

Output:

n	Inverse of n	Sum of terms
---	--------------	--------------

1	1.00e+00	1.00
2	5.00e-01	1.50
3	3.33e-01	1.83
4	2.50e-01	2.08
5	2.00e-01	2.28
6	1.67e-01	2.45
7	1.43e-01	2.59
8	1.25e-01	2.72
9	1.11e-01	2.83

Designing Our Own Manipulators:

- Own manipulators are designed for certain special purposes.
- The general form is

```
ostream &manipulator-name(ostream &output)
{
    .....
    .....(code)
}
```

Example:

```
ostream &unit(ostream &output)
{
    output<<" inches";
    return output;
}
```

Program:

```
#include<iostream.h>
#include<iomanip.h>
ostream &unit(ostream &output)
{
    output<<" inches";
    return output;
}

void main()
{
    cout<<"Height : 56 "<<unit;
}
```

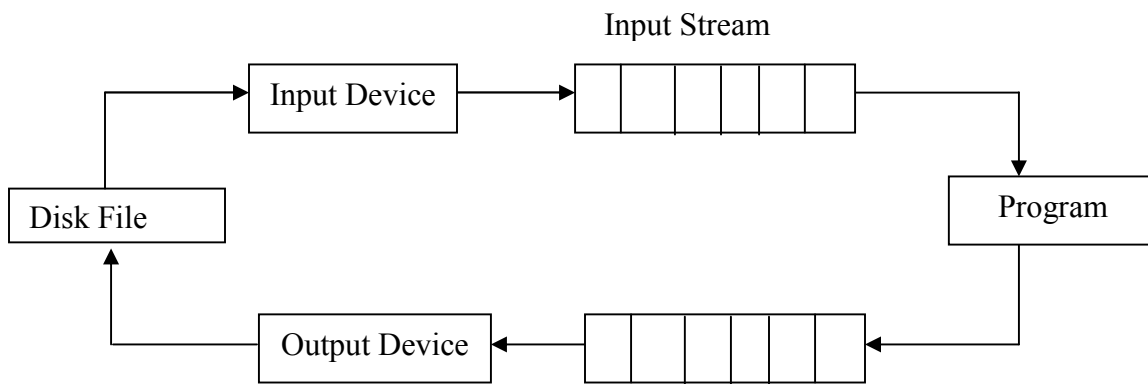
Output:Height :

56 inches

Files: Classes for file stream operations

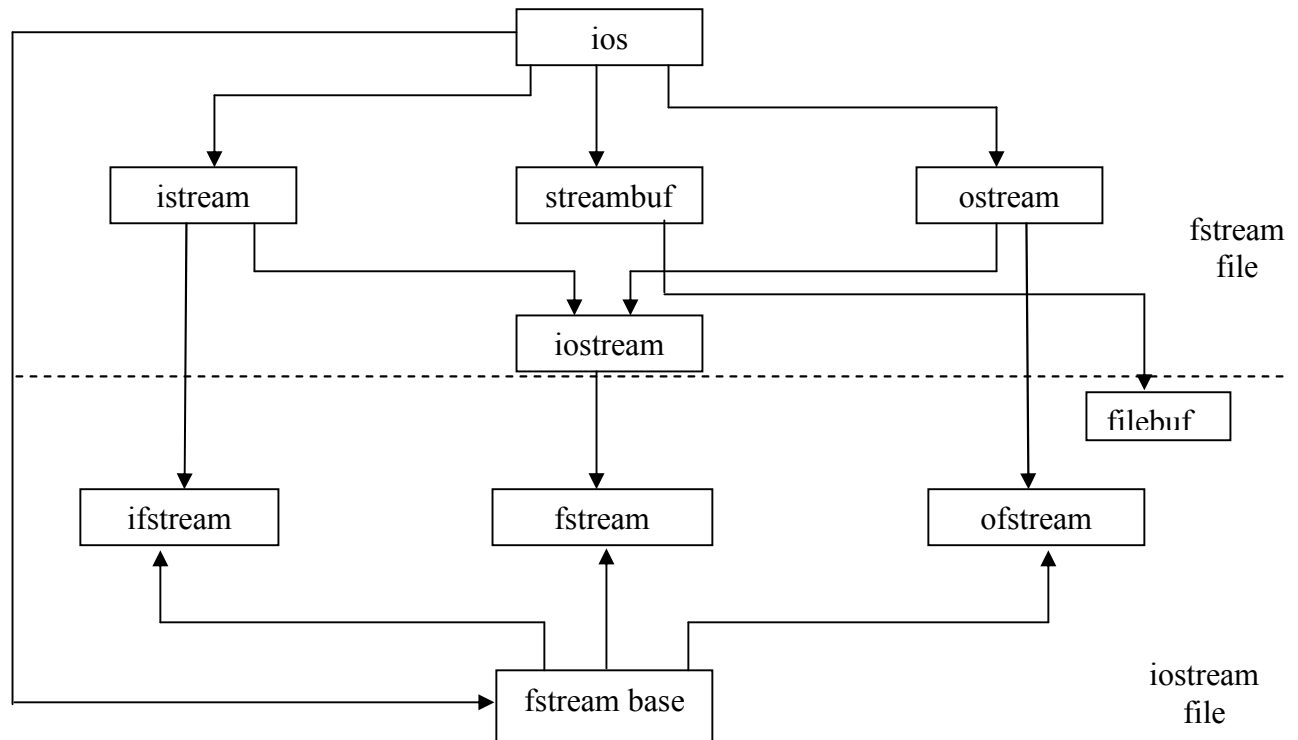
Working With Files:

- The data is stored in secondary devices using the concept of File.
- A File is a collection of related data stored in a particular area on the disk.
- Programs typically involves either or both of the following kinds of data communication:
 - ❖ Data transfer between the console unit and the program.
 - ❖ Data transfer between the program and the disk



Classes for File Stream Operations: Output Stream

- The C++ I/O system contains a set of classes that define the file handling methods.
- File handling class includes ifstream, ofstream, and fstream. These classes are derived from the corresponding iostream class.
- These are the class designed to manage the disk files
- All the classes are declared in fstream so all the program should include this header file



File Operations:

- ❖ Open file
- ❖ Read and Write Operations
- ❖ Closing a file

Opening and closing a file

- Use a disk file requires
- ❖ Suitable name for the file.
- ❖ Data type and structure.
- ❖ Purpose
- ❖ Opening method

Opening Files Using Constructor:

- A constructor is used to initialize an object while it is being created.

- A file name is used to initialize the file stream object.

Steps for Creating Object:

- Create a file stream object to manage the stream using appropriate class. The class ofstream is used to create the output stream and the class ifstream to create the input stream
- Initialize the file object with the desired filename.

Example:

```
ofstream outfile("result");
```

- This create outfile as ofstream object that manages the output stream. This statement also opens the file result and attaches to the output stream outfile.

```
ifstream infile("data")
```

- This create infile as ifstream object that manages the input stream. This statement also opens the file data and attaches to the input stream infile.

Program:

```
#include<iostream.h>
#include<fstream.h>
void main()
{
    ofstream outf("Item");
    char name[30];
    float cost;
    cout<<"Enter the Item Name: ";
    cin>>name;
    outf<<name<<"\n";
    cout<<"Enter the Item Cost: ";
    cin>>cost;
    outf<<cost<<"\n";
```

```

outf.close();
ifstream inf("Item");
inf>>name;
inf>>cost;
cout<<"\n Item Name:"<<name<<"\n";
cout<<"Item cost:"<<cost<<"\n";
inf.close();
}

```

Opening a Files Using open():

➤ The function open() can be used to open multiple files that use the same stream object.

Syntax:

```

File-stream-class stream-object;
stream-object.open("file name");

```

Example:

```

ofstream outfile;
outfile.open("data");
.....
.....
outfile.close();

```

Program:

```

#include<iostream.h>
#include<fstream.h>
void main()
{
    ofstream outf;
    char name[30];
    int i;
    outf.open("prog");

```

```

cout<<"Enter the 3 Programming Language:\n ";
for(i=0;i<3;i++)
{
    cin>>name;
    outf<<name<<"\n";
}
outf.close();
outf.open("soft");
cout<<"Enter the 3 softwares:\n ";
for(i=0;i<3;i++)
{
    cin>>name;
    outf<<name<<"\n";
}
outf.close();
ifstream inf;
inf.open("prog");
cout<<"Programming Language:\n";
while(inf)
{
    inf.getline(name,50);
    cout<<name<<"\n";
}
inf.close();
inf.open("soft");
cout<<"Software:\n";
while(inf)
{
    inf.getline(name,50);

```



```

    cout<<name<<"\n";
}
inf.close();
}

```

Detecting End of File:

- eof() function is used to detect end of File.
- It is the member function of ios class.
- It returns a non-zero value if the end-of-file condition is encountered and a zero otherwise.

Example:

```

if(fileobj.eof() !=0)
{ exit(0);}

```

File Modes:

- istream and ostream constructors and function open() to create new files as well as to open the existing files.
- open() method takes two arguments one for file name and other for mode.

Syntax:

Stream-object.open("file-name",mode);

- mode specifies the purpose for which the file is opened.
- The default mode values are:
 - ❖ ios::in for ifstream functions meaning open for reading only.
 - ❖ ios::out for ofstream functions meaning open for writing only.

File Mode Parameters:

Parameter	Meaning
ios::app	Append to end of file
ios::ate	Go to end of the file on opening
ios::binary	Binary file
ios::in	Open file for reading only

<code>ios::nocreate</code>	Open fails if the file does not exist
<code>ios::noreplace</code>	Opens fails if the file already exist
<code>ios::out</code>	Open file for writing only
<code>ios::trunc</code>	Delete the contents of the file if it exists.

- Opening a file in `ios::out` mode also open it in the `ios::trunc` mode by default.
- `ios::app` and `ios::ate` takes to the end of the file when it is opened
- The difference between `ios::app` and `ios::ate` is `ios::app` allows us to add data to the end of the file but `ios::app` mode permits to add data or to modify data anywhere in the file.
- `ios::app` can be used only with the file capable of output.
- Creating a stream using `ifstream` implies input and creating a stream using `ofstream` implies output. So in this cases it is not necessary to provide the mode parameters.
- The `fstream` class does not provide a mode by default and therefore it is necessary to provide the mode explicitly when using an object of `fstream` class.
- The mode can combine two or more parameters using the bitwise OR operator

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

File Pointers and Their Manipulation

All I/O streams objects have, at least, one internal stream pointer: `ifstream`, like `istream`, has a pointer known as the get pointer that points to the element to be read in the next input operation.

`ofstream`, like `ostream`, has a pointer known as the put pointer that points to the location where the next element has to be written.

Finally, `fstream`, inherits both, the `get` and the `put` pointers, from `iostream` (which is itself derived from both `istream` and `ostream`).

File Manipulators

`seekg()` moves get pointer(input) to a specified location

`seekp()` moves put pointer (output) to a specified location

`tellg()` gives the current position of the get pointer

`tellp()` gives the current position of the put pointer

The other prototype for these functions is:

```
seekg(offset, reposition );
```

```
seekp(offset, reposition );
```

The parameter `offset` represents the number of bytes the file pointer is to be moved from the location specified by the parameter `reposition`.

The `reposition` takes one of the following three constants defined in the `ios` class.

`ios::beg`- start of the file

`ios::cur`- current position of the pointer

`ios::end`- end of the file

example: `file.seekg(-10, ios::cur);`

Sequential input and output operations

➤ The file stream support a number of member functions for performing the input and output operations on files.

put() and get() function:

➤ The function `put()` writes a single character to the associated stream.

➤ The function `get()` reads a single character to the associated stream.

Syntax:

`File-object.get(character)`

`File-object.put(character)`

Program:

```
#include<iostream.h>
#include<fstream.h>
#include<string.h>
void main()
{
    fstream file;
    char name[30];
    int i;
    cout<<"Enter Name: ";
    cin>>name;
    int l=strlen(name);
    file.open("text",ios::in | ios::out);
    for(i=0;i<l;i++)
    {
        file.put(name[i]);
    }
    file.seekg(0);
    char c;
    while(file)
    {
        file.get(c);
        cout<<c;
    }
}
```

```
file.close();
}
```

write() and read() function:

- The function write() and read() handles the data in binary form. This means that the values stored in the disk file in the same format in which they stored in the internal memory.
- An int takes two bytes to store its value in the binary form, irrespective of its size.
- The binary format is more accurate for storing the numbers in the exact internal representation.
- The binary format is much faster to saving the data to.

Syntax:

- ```
inFile-object.read((char *) &v, sizeof(v))
outFile-object.write((char *) &v, sizeof(v))
```
- The first argument is the address of variable v.
  - The second argument is the length of the variable in bytes.
  - The address of the variable must be cast to type char \*.

**Program:**

```
#include<iostream.h>
#include<fstream.h>
#include<iomanip.h>
void main()
{
 float height[5]={176,182,167.89,177.9,160.24};
 ofstream ofile;
 int i;
 ofile.open("data");
 ofile.write((char *) &height, sizeof(height));
 ofile.close();
}
```

```

ifstream infile;
infile.open("data");
infile.read((char *) &height, sizeof(height));
for(i=0;i<5;i++)
{
 cout.setf(ios::showpoint);
 cout<<setw(10)<<setprecision(2)<<height[i]<<endl;
}
infile.close();
}

```

### **Reading and Writing a Class Object:**

- C++ supports features for writing to and reading from the disk files objects directly.
- The binary input and output functions read() and write() are designed to do exactly this job.
- These functions handle the entire structure of an object as a single unit, using the computer's internal representation of data.
- For instance, the function write() copies a class object from the memory byte by byte with no conversion.
- Only data members are written to the disk file and the member functions are not.
- The length of the object is obtained by sizeof operator.

### **Program:**

```

#include<iostream.h>
#include<fstream.h>
#include<iomanip.h>
class Inventory
{
 char name[20];

```

```
int code;
float cost;
public:
 void readdata();
 void show();
};
void Inventory::readdata()
{
 cout<<"Enter Name: ";
 cin>>name;
 cout<<"Enter Code: ";
 cin>>code;
 cout<<"Enter Cost: ";
 cin>>cost;
}
void Inventory::show()
{
 cout<<setiosflags(ios::left)<<setw(10)<<name
 <<setiosflags(ios::right)<<setw(10)<<code
 <<setprecision(2)<<setw(10)<<cost<<endl;
}
void main()
{
 Inventory item[3];
 fstream file;
 file.open("stock.dat",ios::in | ios::out);
 cout<<"Enter Details of Items\n";
 for(int i=0;i<3;i++)
 {
```

```

 item[i].readdata();
 file.write((char *) &item[i], sizeof(item[i]));
}
file.seekg(0);
cout<<"\n\nOutput\n\n";
for(i=0;i<3;i++)
{
 file.read((char *) &item[i], sizeof(item[i]));
 item[i].show();
}
file.close();
}

```

### **updating a file random access**

- Updating is a routine take in the maintenance of any data file.
- Updating include the following task.
- ❖ Displaying the contents of a file.
- ❖ Modifying an existing item.
- ❖ Adding a new item.
- ❖ Deleting an existing item.

#### **Program:**

```

#include<iostream.h>
#include<fstream.h>
#include<iomanip.h>
class Inventory
{
 char name[20];
 int code;
 float cost;

```



```
public:
 void readdata();
 void show();
};

void Inventory::readdata()
{
 cout<<"Enter Name: ";
 cin>>name;
 cout<<"Enter Code: ";
 cin>>code;
 cout<<"Enter Cost: ";
 cin>>cost;
}

void Inventory::show()
{
 cout<<setiosflags(ios::left)<<setw(10)<<name
 <<setiosflags(ios::right)<<setw(10)<<code
 <<setprecision(2)<<setw(10)<<cost<<endl;
}

void main()
{
 Inventory item;
 fstream file;
 file.open("stock.dat",ios::ate | ios::in | ios::out | ios::binary);
 file.seekg(0,ios::beg);
 cout<<"\nCurrent Contant of File\n";
 while(file.read((char *) &item, sizeof(item)))
 {
 item.show();
 }
}
```

```
}
file.clear();
cout<<"\nAdd An Item\n";
item.readdata();
char ch;
cin.get(ch);
file.write((char *) &item, sizeof(item));
file.seekg(0);
cout<<"\nContant of File After Appended\n";
while(file.read((char *) &item, sizeof(item)))
{
 item.show();
}
int ls=file.tellg();
int n=ls/sizeof(item);
cout<<"\nNumber of Objects="<<n;
cout<<"\nTotal bytes in the file="<<ls;
cout<<"Modify An Item";
int no;
cout<<"\nEnter the Object Number to Update : ";
cin>>no;
cin.get(ch);
int loc=(no-1)*sizeof(item);
if(file.eof())
 file.clear();
file.seekp(loc);
cout<<"\nEnter New values of object:\n";
item.readdata();
cin.get(ch);
```

```
file.write((char *) &item, sizeof(item))<<flush;
file.seekg(0);
cout<<"\nContent of File After Modified\n";
while(file.read((char *) &item, sizeof(item)))
{
 item.show();
}
file.close();
}
```

### **Command-line Arguments:**

- C++ support a feature of supply of arguments to the main() function.
- The command-line arguments are achieved by the arguments of the main() function.

### **Syntax:**

```
main(int argc, char *argv[])
```

- ❖ argc known as argument counter, represents the number of arguments in the command line.
- ❖ argv known as argument vector, is an array of char type pointers that points to the command line arguments.
- ❖ The size of this array will be equal to the value of argc.
- Arguments are supplied at the time of invoking the program.

### **Example:**

```
C:\>program-file-name first-file second-file
```

- Program-file-name is the name of the file containing the program to be executed.
- first-file and second-file are the file names passed to the program as command-line arguments.

- The first argument is always the file name and contains the program to be executed.
- The value of argc would be 3 and the argv would be an array of 3 pointers to strings
- ❖ argv[0] → program-file-name
- ❖ argv[1] → first-file-name // used for reading purpose
- ❖ argv[2] → second-file-name // used for writing purpose

### **Templates and Exceptions:- Templates**

#### **Templates:**

- Templates is one of the features added to C++ recently.
- It is a new concept which enables us to define generic classes and functions and thus provides support for generic programming.
- Generic programming is an approach where generic types are used as parameters in algorithms so that they work for a variety of suitable data types and data structure.
- A template can be used to create a family of classes or functions.
- For example, a class template for an array class would enable us to create arrays of various data types such as int array and float array.
- Similarly, define a template for a function, say mul(), that would help us create various versions of mul() for multiplying int, float and double type values.
- A template can be considered as a kind of macro.
- When an object of a specific type is defined for actual use, the template definition for that class is substitute with required data type. Since a template defined with a parameter that would be replaced by a specified data type at the time of actual use of the class or function, the templates are sometimes called parameterized classes or functions.

**Class templates****Class Templates:**

- A simple process to create a generic class using a template with anonymous type.
- template is the keyword used to create Template
- The class template definition is very similar to an ordinary class definition expect the prefix template<class T> and the use of type T.
- This prefix tells the compiler that is going to declare a template and use T as a type name in the declaration.

**Syntax:**

```

template <class T>
class class-name
{
 //.....
 //class member specification
 //with anonymous type T
 //wherever appropriate
 //.....
};

```

**Example:**

```

int size=3;
template<class T>
class vector
{
 T* v;
 int size;
 public:
 vector()
 {

```

```

 v=new T[size];
 for(int i=0;i<3;i++)
 v[i]=0;
}
vector(T* a)
{
 for(int i=0;i<size;i++)
 v[i]=a[i];
}
T operator *(vector &y)
{
 T sum=0;
 for(int i=0;i<size;i++)
 sum+=this->v[i]*y.v[i];
 return sum;
}
};

```

### **Class Templates with Multiple Parameters:**

- More than one generic data type in a class template.
- It is declared as a comma separated list within the template specification .

### **Syntax:**

```

template <class T1, class T2,...,class Tn>
class class-name
{

 //body of the class
};

```

**Program:**

```
#include<iostream.h>
template<class T1, class T2>
class Test
{
 T1 a;
 T2 b;
public:
 Test(T1 x, T2 y)
 {
 a=x;
 b=y;
 }
 void show()
 {
 cout<<"\na : "<<a<<"\nb : "<<b;
 }
};
void main()
{
 Test <float, int> t1(1.23,123);
 Test <int, char> t2(100,'M');
 t1.show();
 t2.show();
}
```

**Function templates**

➤ Defining function Templates that could be used to create a family of functions with different argument types.

**Syntax:**

```

template <class T>
return-type function-name(argument of type T)
{
 //.....
 //body of function
 //with type T
 //wherever appropriate
 //.....
}

```

- The function template syntax is similar to that of the class template expect that defining functions instead of classes.
- Use template parameter T as and when necessary in the function body and its argument list.

**Program:**

```

#include<iostream.h>
template<class T>
void swap(T &x, T &y)
{
 T temp=x;
 x=y;
 y=temp;
}
void fun(int m,int n,float a,float b)
{
 cout<<"\nm and n before swap: "<<m<<" "<<n;
 swap(m,n);
 cout<<"\nm and n after swap: "<<m<<" "<<n;
 cout<<"\na and b before swap: "<<a<<" "<<b;
}

```



```

swap(a,b);
cout<<"\na and b after swap: "<<a<<" "<<b;
}
void main()
{
 fun(100,200,11.53,33.44);
}

```

### **Function Templates with Multiple Parameters:**

➤ Use more than one generic data type in the template statement using a comma-separated list.

#### **Syntax:**

```

template <class T1, class T2,...,class Tn>
return-type function-name(arguments of types T1,T2,...)
{

 //body of the function
}

```

#### **Program:**

### **Overloading of Template Functions:**

- A template function may be overloaded either by template functions or ordinary functions of its name.
- The overloading resolution is accomplished as follows:
  - ❖ Call an ordinary function that has an exact match.
  - ❖ Call a template function that could be created with an exact match.
  - ❖ Try normal overloading resolution to ordinary functions and call the one that matches.

- An error is generated if no match is found.
- No automatic conversions are applied to arguments on the template functions.

**Program:**

```
#include<iostream.h>
template<class T>
void display(T x)
{
 cout<<"\nTemplate method : "<<x;
}
void display(int x)
{
 cout<<"\nExplicit method : "<<x;
}

void main()
{
 display(11.53);
 display(44);
 display("welcome");
}
```

**Member function templates**

- All the member functions were defined as inline is not necessary.
- Define members outside that class is also possible.
- The member function of the template classes are parameterized by the type arguments and functions must be defined by the function templates.

**Syntax:**

```

template <class T>
return-type class-name<T>:: function-name(argument list)
{

 //body of the function
}

```

**Example:**

```

template<class T>
class vector
{
 T* v;
 int size=3;
 public:
 vector(int m);
 vector(T* a);
 T operator*(vector &y);
};

template<class T>
vector<T>::vector(int m)
{
 v=new T[size];
 for(int i=0;i<size;i++)
 v[i]=0;
}

template<class T>
vector<T>::vector(T* a)
{
 for(int i=0;i<size;i++)

```

```

 v[i]=a[i];
 }
 template<class T>
 vector<T>::operator *(vector &y)
 {
 T sum=0;
 for(int i=0;i<size;i++)
 sum+=this->v[i]*y.v[i];
 return sum;
 }

```

### **Exception handling**

Exceptions are run-time anomalies, such as division by zero, that require immediate handling when encountered by your program. The C++ language provides built-in support for raising and handling exceptions. With C++ exception handling, your program can communicate unexpected events to a higher execution context that is better able to recover from such abnormal events. These exceptions are handled by code that is outside the normal flow of control

The C++ language provides built-in support for handling anomalous situations, known as exceptions, which may occur during the execution of your program. The try, throw, and catch statements implement exception handling. With C++ exception handling, your program can communicate unexpected events to a higher execution context that is better able to recover from such abnormal events. These exceptions are handled by code that is outside the normal flow of control. The Microsoft

C++ compiler implements the C++ exception handling model based on the ANSI C++ standard.

The following syntax shows a try block and its handlers:

```
try {
 // code that could throw an exception
}
[catch (exception-declaration) {
 // code that executes when exception-declaration is thrown
 // in the try block
}
[catch (exception-declaration) {
 // code that handles another exception type
}] ...]
// The following syntax shows a throw expression:
throw [expression]
```

C++ also provides a way to explicitly specify whether a function can throw exceptions. You can use exception specifications in function declarations to indicate that a function can throw an exception. For example, an exception specification `throw(...)` tells the compiler that a function can throw an exception, but doesn't specify the type, as in this example:

```
void MyFunc() throw(...) {
 throw 1;
}
```

---

**Sample Programs in c++**

## Prime Number

```
#include<iostream.h>
#include<conio.h>
int main()
{
clrscr();
int st_no,end_no,div,no_div=0;
cout<<"Enter the starting no ";
cin>>st_no ;
cout<<"Enter the enging no ";
cin>>end_no ;

 while(st_no<=end_no)
 { div=st_no;
 no_div=0;

 while(div>=1)
 {
 if(st_no%div==0)
 {
 no_div= no_div+ 1 ;
 }
 div--;
 }

 if(no_div<=2)
 {
 cout<<st_no<<" IS PRIME"<<endl<<endl;
```

```

}
st_no++;
}
return 0;
}

```

#### String Program

```

#include <iostream>
#include <string>
#include <fstream>
#include <conio.h>
using namespace std;
int main()
{
 ifstream file;
 string s, city, bigstring, substring;
 int count = 0;
 file.open("c:\\cities.txt");
 cout << "Enter all or part of a city name: ";
 getline(cin,city);
 if (bigstring.find(substring) != -1)
 while (getline(file,s))
 {
 cout << s << endl;
 count++
 }
 cout << "There were " << count << " matches in the file" << endl;
 getch();
 return 0;
}

```

## Palindrome

```
#include <iostream>
#include <deque>
#include <string>
#include <cctype>
using namespace std;
int main()
{
 string input;
 deque<string> stackOne;
 deque<string> stackTwo;
 cout << "Enter a text . Do not include spaces or punctuation.\n";
 getline(cin, input);
 stackOne.push_front(input);
 while(!stackOne.empty())
 {
 input = stackOne.front();//retrieve the user entered input
 stackTwo.push_front(input); // put input into stack two at
the front
 }
 if(stackOne == stackTwo)
 {
 cout << "It is a palindrome." << endl;
 }
 else
 cout << "It is not a palindrome." << endl;
 return 0;
}
```





**KARPAGAM ACADEMY OF HIGHER EDUCATION**  
(Deemed University Established Under Section 3 of UGC Act 1956)  
Coimbatore – 21  
(For the candidates admitted from 2016 onwards)  
**DEPARTMENT OF COMMERCE (CA)**

**SUBJECT : OBJECT ORIENTED PROGRAMMING WITH C++**

**SEMESTER : III**

**SUBJECT CODE: 16CCU302**

**CLASS : II B.COM CA**

**POSSIBLE QUESTIONS – UNIT V**

**PART A (1 Mark)**

**(Online Examinations)**

**PART B (2 Marks)**

1. Define Pointer to object
2. Define to Derived classes
3. What is Virtual Functions
4. How pointer to derived Classes used in a program
5. Define File
6. List out File Stream Operation'
7. How will you Open and Close a file
8. What is File Pointer
9. What are Sequential I/O operations
10. How files are manipulated.
11. What are streams?

**PART C ( 6 Marks)**

1. Describe on file operations.
2. Explain Virtual Functions with example.
3. Describe about Pointers to Object
4. Describe about Pointers to Derived Classes.
5. Explain file stream Operations.
6. Explain file pointers and their manipulators.
7. List out and explain Sequential I/O operations.
8. Write a program using file operation (to open and close a file).
9. Describe file pointers with example.
10. Differentiate between pointers to object and pointers to derived class.