

SCOPE:

This course provides student with a comprehensive study of the fundamentals of C and C++ programming language. Classroom lectures stress the strength of C, which provide programmers with the means of writing efficient, maintainable and portable code.

OBJECTIVES:

- Know the basic concept of computers
- Understand the concept of a program (i.e., a computer following a series of instructions)
- Understand the concept of a loop – that is, a series of statements which is written once but executed repeatedly- and how to use it in a programming language
- Be able to break a large problem into smaller parts, writing each part as a module or function
- Understand the concept of a program in a high-level language being translated by a compiler into machine language program and then executed.

UNIT I**Introduction to C and C++:**

History of C and C++, Overview of Procedural Programming and Object-Orientation Programming, Using main() function, Compiling and Executing Simple Programs in C++.

Data Types, Variables, Constants, Operators and Basic I/O: Declaring, Defining and Initializing Variables, Scope of Variables, Using Named Constants, Keywords, Data Types, Casting of Data Types, Operators (Arithmetic, Logical and Bitwise), Using Comments in programs, Character I/O (getc, getchar, putc, putchar etc), Formatted and Console I/O (printf(), scanf(), cin, cout), Using Basic Header Files (stdio.h, iostream.h, conio.h etc). **Expressions, Conditional Statements and Iterative Statements:** Simple Expressions in C++ (including Unary Operator Expressions, Binary Operator Expressions), Understanding Operators Precedence in Expressions, Conditional Statements (if construct, switch-case construct), Understanding syntax and utility of Iterative Statements (while, do-while, and for loops), Use of break and continue in Loops, Using Nested Statements (Conditional as well as Iterative)

UNIT II

Functions and Arrays: Utility of functions, Call by Value, Call by Reference, Functions returning value, Void functions, Inline Functions, Return data type of functions, Functions parameters, Differentiating between Declaration and Definition of Functions, Command Line Arguments/Parameters in Functions, Functions with variable number of Arguments.

Creating and Using One Dimensional Arrays (Declaring and Defining an Array, Initializing an Array, Accessing individual elements in an Array, Manipulating array elements using loops), Use Various types of arrays (integer, float and character arrays / Strings) Two-dimensional Arrays (Declaring, Defining and Initializing Two Dimensional Array, Working with Rows and Columns), Introduction to Multi-dimensional arrays.

UNIT III

Derived Data Types (Structures and Unions): Understanding utility of structures and unions, Declaring, initializing and using simple structures and unions, Manipulating individual members of structures and unions, Array of Structures, Individual data members as structures, Passing and returning structures from functions, Structure with union as members, Union with structures as members. **Pointers and References in C++:** Understanding a Pointer Variable, Simple use of Pointers (Declaring and Dereferencing Pointers to simple variables), Pointers to Pointers, Pointers to structures, Problems with Pointers, Passing pointers as function arguments, Returning a pointer from a function, using arrays as pointers, Passing arrays to functions. Pointers vs. References, Declaring and initializing references, using references as function arguments and function return values

UNIT IV

Memory Allocation in C++: Differentiating between static and dynamic memory allocation, use of malloc, calloc and free functions, use of new and delete operators, storage of variables in static and dynamic memory allocation. **File I/O, Preprocessor Directives:** Opening and closing a file (use of fstream header file, ifstream, ofstream and fstream classes), Reading and writing Text Files, Using put(), get(), read() and write() functions, Random access in files, Understanding the Preprocessor Directives (#include, #define, #error, #if, #else, #elif, #endif, #ifdef, #ifndef and #undef), Macros.

UNIT V

Using Classes in C++: Principles of Object-Oriented Programming, Defining & Using Classes, Class Constructors, Constructor Overloading, Function overloading in classes, Class Variables & Functions, Objects as parameters, Specifying the Protected and Private

ACSUess, Copy Constructors, Overview of Template classes and their use.**Overview of Function Overloading and Operator Overloading:** Need of Overloading functions and operators, Overloading functions by number and type of arguments, Looking at an operator as a function call, Overloading Operators (including assignment operators, unary operators)**Inheritance, Polymorphism and Exception Handling:** Introduction to Inheritance (Multi-Level Inheritance, Multiple Inheritance), Polymorphism (Virtual Functions, Pure Virtual Functions), Basics Exceptional Handling (using catch and throw, multiple catch statements), Catching all exceptions, Restricting exceptions, Rethrowing exceptions.

SUGGESTED READINGS

1. HerbtzSchildt, (2003). C++: The Complete Reference, Fourth Edition, McGraw Hill.
2. BjarneStroustrup, (2013). The C++ Programming Language, Fourth Edition, Addison-Wesley.
3. BjarneStroustrup, (2014).Programming Principles and Practice using C++, Second Edition, Addison-Wesley.
4. E Balaguruswamy,(2008). Object Oriented Programming with C++, Tata McGraw-Hill Education.New Delhi.
5. Paul Deitel, Harvey Deitel, (2011) .C++ How to Program, Eighth Edition, Prentice Hall.
6. John R. Hubbard, (2000).Programming with C++, Schaum's Series, Second Edition. McGraw Hill Professional.
7. Andrew Koeni, Barbara, E. Moo,(2000). ACSUelated C++, Published by Addison Wesley .
8. Scott Meyers,(2005). Effective C++, Third Edition, Published by Addison-Wesley.
9. Harry, H. Chaudhary, (2014).Head First C++ Programming: The Definitive Beginner's Guide, FirstCreate spaceInc, O-D Publishing, LLC USA.
10. Walter Savitch, (2007). Problem Solving with C++, Pearson Education.
11. Stanley B. Lippman, JoseeLajoie., Barbara E. Moo.,(2012). C++ Primer, Published by Addison- Wesley, 5th Edition.

WEB SITES

1. <http://www.cs.cf.ac.uk/Dave/C/CE.html>
2. <http://www2.its.strath.ac.uk/courses/c/>
3. <http://www.iu.hio.no/~mark/CTutorial/CTutorial.html>
4. <http://www.cplusplus.com/doc/tutorial/>
5. www.cplusplus.com/
6. www.cppreference.com/

KARPAGAM ACADEMY OF HIGHER EDUCATION

(Established Under Section 3 of UGC Act 1956)

COIMBATORE – 641 021

DEPARTMENT OF COMPUTER APPLICATIONS

Batch: 2016-2019

16MMU304B PROGRAMMING WITH C and C++

LECTURE PLAN

UNIT – I

SL. NO	LECTURE DURATION (HR)	TOPICS TO BE COVERED	SUPPORT MATERIALS
1.	1	History of C and C++, Overview of Procedural Programming and Object-Orientation Programming	W1; T4: 1 -14
2.	1	Using main() function, Compiling and Executing Simple Programs in C++.	W1; T4: 19 – 22, 26 - 30
3.	1	Data Types, Variables, Constants, Operators and Basic I/O: Declaring, Defining and Initializing Variables, Scope of Variables, Using Named Constants, Keywords	W1; T4: 35-45, T1: 34 - 52
4.	1	Data Types, Casting of Data Types	T4: 46-57
5.	1	Operators (Arithmetic, Logical and Bitwise)	T1: 13 - 33
6.	1	Using Comments in programs	T4: 23 - 25, T1: 187 – 210,250 - 254
7.	1	Character I/O (getc, getchar, putc, putchar etc)	T4: 23 - 25, T1: 187 – 210,250 - 254
8.	1	Formatted and Console I/O (printf(), scanf(), cin, cout),	T4: 23 - 25, T1: 187 – 210,250 - 254
9.	1	Using Basic Header Files (stdio.h, iostream.h, conio.h etc).	T4: 23 - 25, T1: 187 – 210,250 - 254
10.	1	Expressions, Conditional Statements and Iterative Statements: Simple Expressions in C++ (including Unary Operator Expressions)	T4: 58 – 60, T1: 53 - 56
11.	1	Binary Operator Expressions	T4: 58 – 60, T1: 53 - 56
12.	1	Understanding Operators Precedence in Expressions	T4: 58 – 60, T1: 53 - 56
13.	1	Conditional Statements (if construct, switch-case construct)	T4: 64 – 69, T1: 59 – 70
14.	1	Understanding syntax and utility of Iterative Statements (while, do-while, and for loops),	T4: 64– 69, T1: 70 – 79
15.	1	Understanding syntax and utility of Iterative Statements (while, do-while, and for loops),	T4: 64– 69, T1: 70 – 79
16.	1	Use of break and continue in Loops	T1: 80 – 88

17.	1	Using Nested Statements (Conditional as well as Iterative)	T1: 80 – 88
18.	1	Recapitulation and Discussion on Important Questions	

Text Books:

T1 - HerbtzSchildt, "C++: The Complete Reference", Fourth Edition, McGraw Hill.2003

T4 - E Balaguruswamy, "Object Oriented Programming with C++", Tata McGraw-Hill Education, 2008.

WEB SITES

W1 -<http://www.cs.cf.ac.uk/Dave/C/CE.html>

UNIT – II

Sl. NO	LECTURE DURATION (HR)	TOPICS TO BE COVERED	SUPPORT MATERIALS
1.	1	Functions and Arrays: Utility of functions	T4 : 77 – 82, T1 : 137 - 142
2.	1	Call by Value, Call by Reference, Return data type of functions	T4 : 77 – 82, T1 : 137 - 142
3.	1	Functions returning value	T4 : 82 - 84, T6 : 99, 109
4.	1	Void functions, Inline Functions	T4 : 82 - 84, T6 : 99, 109
5.	1	Functions parameters	T4 : 82 - 84, T6 : 99, 109
6.	1	Differentiating between Declaration and Definition of Functions	T4 : 82 - 84, T6 : 99, 109
7.	1	Command Line Arguments	T4 :85 – 90, T1: 144 - 147
8.	1	Parameters in Functions	T4 :85 – 90, T1: 144 - 147
9.	1	Functions with variable number of Arguments.	T4 : 769 - 770
10.	1	Creating and Using One Dimensional Arrays (Declaring and Defining an Array, Initializing an Array)	T1 : 89 – 92, T5 : 283 – 285
11.	1	Creating and Using One Dimensional Arrays (Declaring and Defining an Array, Initializing an Array)	T1 : 89 – 92, T5 : 283 – 285
12.	1	Accessing individual elements in an Array	T1 : 92 - 95, T5 : 286 – 298
13.	1	Manipulating array elements using loops	T1 : 92 - 95, T5 : 286 – 298
14.	1	Use Various types of arrays (integer, float and character arrays / Strings)	T5 : 299 – 315
15.	1	Two-dimensional Arrays (Declaring, Defining and Initializing Two Dimensional Array, Working with Rows and Columns),	T1 : 96 – 100
16.	1	Two-dimensional Arrays (Declaring, Defining and Initializing Two Dimensional Array, Working with Rows and Columns),	T1 : 96 – 100
17.	1	Introduction to Multi-dimensional arrays.	T1 : 101 - 108
18.	1	Recapitulation and Discussion on Important Questions	

Text Books:

T1 - HerbtzSchildt, "C++: The Complete Reference", Fourth Edition, McGraw Hill.2003

T3 - Harry, H. Chaudhary, "Head First C++ Programming: The Definitive Beginner's Guide", First Create space Inc, O-D Publishing, LLC USA.2014

T4 - E Balaguruswamy, "Object Oriented Programming with C++", Tata McGraw-Hill Education, 2008.

T5 - Paul Deitel, Harvey Deitel, "C++ How to Program", 8th Edition, Prentice Hall, 2011.

UNIT – III

SL. NO	LECTURE DURATION (HR)	TOPICS TO BE COVERED	SUPPORT MATERIALS
1.	1	Derived Data Types (Structures and Unions): Understanding utility of structures and unions	T1 : 161 – 163, T2 : 201 – 224
2.	1	Declaring, initializing and using simple structures and unions	T1 : 164 – 176, T2 : 201 – 224
3.	1	Manipulating individual members of structures and unions, Array of Structures	T1 : 176 – 180, T2 : 201 – 224
4.	1	Individual data members as structures, Passing and returning structures from functions	T1 : 180 – 184, T2 : 201 – 224
5.	1	Structure with union as members, Union with structures as members.	T1 : 184 – 186, T2 : 201 – 224
6.	1	Pointers and References in C++: Understanding a Pointer Variable	T1: 113 -120
7.	1	Simple use of Pointers (Declaring and Dereferencing Pointers to simple variables)	T1: 113 -120
8.	1	Pointers to Pointers	T1: 121 – 130
9.	1	Pointers to structures	T1: 121 – 130
10.	1	Problems with Pointers	T1: 121 – 130
11.	1	Passing pointers as function arguments	T1: 131 – 136
12.	1	Returning a pointer from a function	T1: 131 – 136
13.	1	Using arrays as pointers	T1: 131 – 136
14.	1	Passing arrays to functions.	T1: 131 – 136
15.	1	Pointers vs. References	T1: 321 – 349
16.	1	Declaring and initializing references	T1: 321 – 349
17.	1	Using references as function arguments and function return values	T6: 157 – 184, J3
18.	1	Recapitulation and Discussion on Important Questions	

Text Books:

T1 - HerbtzSchildt, "C++: The Complete Reference", Fourth Edition, McGraw Hill.2003

T2 - BjarneStroustrup, "The C++ Programming Language", 4th Edition, Addison-Wesley , 2013.

T6 - John R. Hubbard, "Programming with C++", Schaum's Series, 2nd Edition, 2000.

JOURNALS

J3: IEEE transactions on software engineering

UNIT – IV

SL. NO	LECTURE DURATION (HR)	TOPICS TO BE COVERED	SUPPORT MATERIALS
1.	1	Memory Allocation in C++: Differentiating between static and dynamic memory allocation	T1 : 349 – 350, J1
2.	1	use of malloc, calloc and free functions	T1: 753 -756
3.	1	use of new and delete operators	T1: 351 - 359
4.	1	storage of variables in static and dynamic memory allocation	T4: 114 - 118
5.	1	File I/O, Preprocessor Directives: Opening and closing a file (use of fstream header file, ifstream, ofstream and fstream classes),	T1: 211 -215
6.	1	File I/O, Preprocessor Directives: Opening and closing a file (use of fstream header file, ifstream, ofstream and fstream classes),	T1: 211 -215
7.	1	File I/O, Preprocessor Directives: Opening and closing a file (use of fstream header file, ifstream, ofstream and fstream classes),	T1: 211 -215
8.	1	Reading and writing Text Files	T1: 216 -218
9.		Using put(), get()	
10.	1	Using read() and write() functions,	T1: 218 - 235
11.	1	Random access in files	T1: 559 - 563
12.	1	Reading and writing Text Files	T1: 216 -218
13.	1	Using put(), get(), read() and write() functions,	T1: 218 - 235
14.	1	Random access in files	T1: 559 - 563
15.	1	Understanding the Preprocessor Directives (#include, #define, #error, #if, #else, #elif, #endif, #ifdef, #ifndef and #undef)	T1: 241 - 249
16.	1	Understanding the Preprocessor Directives (#include, #define, #error, #if, #else, #elif, #endif, #ifdef, #ifndef and #undef)	T1: 241 - 249
17.	1	Macros	T1: 240, 250
18.	1	Recapitulation and Discussion on Important Questions	

Text Books:

T1 - HerbtzSchildt, "C++: The Complete Reference", Fourth Edition, McGraw Hill.2003

T4 - E Balaguruswamy, "Object Oriented Programming with C++", Tata McGraw-Hill Education, 2008.

JOURNALS

J1: Computing in science & engineering

UNIT – V

SL NO	LECTURE DURATION (HR)	TOPICS TO BE COVERED	SUPPORT MATERIALS
1.	1	Using Classes in C++: Principles of Object-Oriented Programming	T3 : 55 -84, J1
2.	1	Defining & Using Classes, Class Constructors, Copy Constructors	T4: 144 -164
3.	1	Constructor Overloading, Function overloading in classes	T1: 361 – 367
4.	1	Class Variables & Functions, Objects as parameters, Specifying the Protected and Private Access	T1: 290 – 293
5.	1	Overview of Template classes and their use.	T1 : 461 – 487
6.	1	Overview of Function Overloading and Operator Overloading: Need of Overloading functions and operators	T1 : 380 – 416
7.	1	Overloading functions by number and type of arguments	T1: 393 – 398
8.	1	Looking at an operator as a function call	T1: 393 – 398
9.	1	Overloading Operators (including assignment operators, unary operators)	T4: 171 – 200
10.	1	Inheritance, Polymorphism and Exception Handling: Introduction to Inheritance	T4 : 201 – 248, T1: 419 – 439
11.	1	Multi-Level Inheritance, Multiple Inheritance	T4 : 201 – 248, T1: 419 – 439
12.	1	Polymorphism (Virtual Functions, Pure Virtual Functions),	T1: 440 – 460
13.	1	Basics Exceptional Handling (using catch and throw, multiple catch statements),	T1: 489 – 510, T4: 380 – 385
14.	1	Catching all exceptions, Restricting exceptions, Rethrowing exceptions.	T4: 386 – 400
15.	1	Recapitulation and Discussion on Important Questions	
16.	1	Summarization and Discussion of previous ESE Questions	
17.	1	Summarization and Discussion of previous ESE Questions	
18.	1	Discussion of previous ESE Questions	

Text Books:

T1 - HerbtzSchildt, "C++: The Complete Reference", Fourth Edition, McGraw Hill.2003

T2 - BjarneStroustrup, "The C++ Programming Language", 4th Edition, Addison-Wesley , 2013.

T3 - Harry, H. Chaudhary, "Head First C++ Programming: The Definitive Beginner's Guide", First Create space Inc, O-D Publishing, LLC USA.2014

T4 - E Balaguruswamy, "Object Oriented Programming with C++", Tata McGraw-Hill Education, 2008.

T5 - Paul Deitel, Harvey Deitel, "C++ How to Program", 8th Edition, Prentice Hall, 2011.

T6 - John R. Hubbard, "Programming with C++", Schaum's Series, 2nd Edition, 2000.

WEB SITES

W1 -<http://www.cs.cf.ac.uk/Dave/C/CE.html>

W2 - www.cplusplus.com/

W3 - www.cppreference.com/

JOURNALS

J1: Computing in science & engineering

J2: Science of computer programming

J3: IEEE transactions on software engineering

KARPAGAM ACADEMY OF HIGHER EDUCATION

(Established Under Section 3 of UGC Act 1956)

COIMBATORE – 641 021

DEPARTMENT OF COMPUTER APPLICATIONS

Batch 2016-2019

Subject : PROGRAMMING WITH C and C++

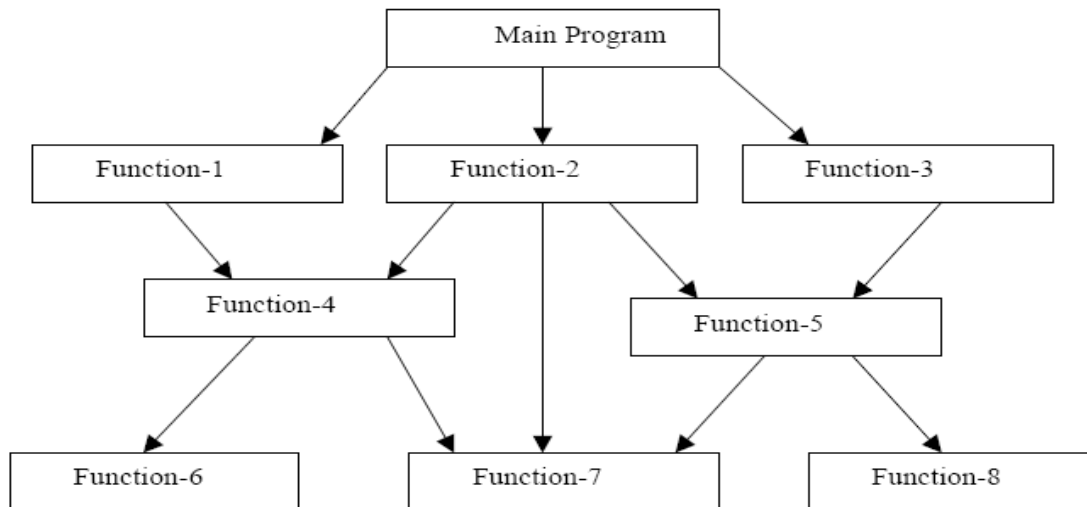
Sub.code : 16MMU304B

History of C / C++

C++ is an object-oriented programming language. It was developed by Bjarne Stroustrup at AT&T Bell Laboratories in Murray Hill, New Jersey, USA, in the early 1980's. Stroustrup, an admirer of Simula67 and a strong supporter of C, wanted to combine the best of both the languages and create a more powerful language that could support object-oriented programming features and still retain the power and elegance of C. The result was C++. Therefore, C++ is an extension of C with a major addition of the class construct feature of Simula67. Since the class was a major addition to the original C language, Stroustrup initially called the new language 'C with classes'. However, later in 1983, the name was changed to C++. The idea of C++ comes from the C increment operator ++, thereby suggesting that C++ is an augmented version of C. C++ is a superset of C. Almost all C programs are also C++ programs. However, there are a few minor differences that will prevent a C program to run under C++ compiler. We shall see these differences later as and when they are encountered. The most important facilities that C++ adds on to C are classes, inheritance, function overloading and operator overloading. These features enable creating of abstract data types, inherit properties from existing data types and support polymorphism, thereby making C++ a truly object-oriented language.

Overview of Procedural Programming and Object-Oriented Programming**Procedure-Oriented Programming**

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



Procedure oriented programming basically consists of writing a list of instructions for the computer to follow, and organizing these instructions into groups known as *functions*. We normally use flowcharts to organize these actions and represent the flow of control from one action to another.

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

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

Some Characteristics exhibited by procedure-oriented programming are:

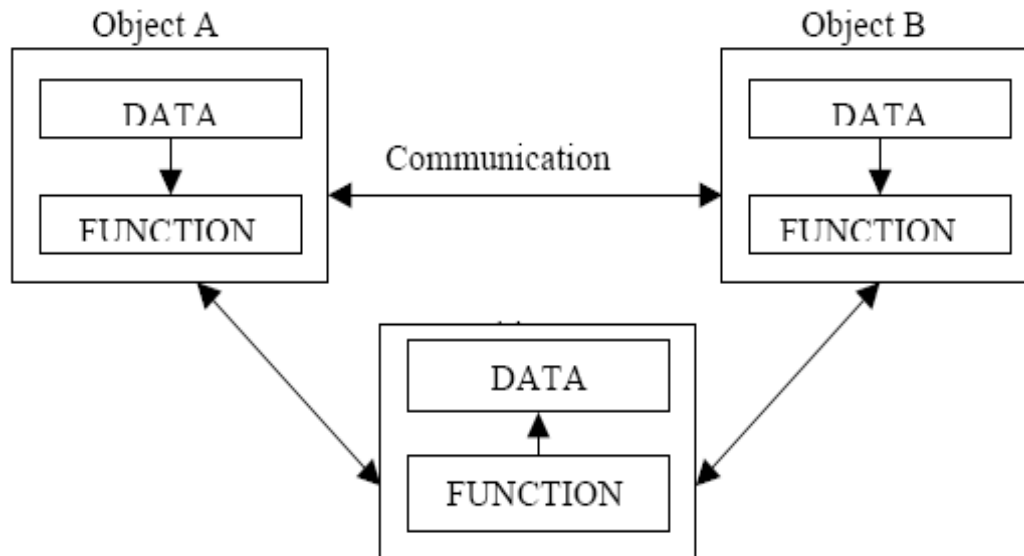
- Emphasis is on doing things (algorithms).
- Large programs are divided into smaller programs known as functions.
- Most of the functions share global data.
- Data move openly around the system from function to function.
- Functions transform data from one form to another.
- Employs top-down approach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

Object Oriented Paradigm

The major motivating factor in the invention of object-oriented approach is to remove some of the flaws encountered in the procedural approach. OOP treats data as a critical element in the program development and does not allow it to flow freely around the system. It ties data more closely to the function that operate on it, and protects it from accidental modification

from outside function. OOP allows decomposition of a problem into a number of entities called objects and then builds data and function around these objects. The organization of data and function in object-oriented programs is shown in figure below. The data of an object can be accessed only by the function associated with that object. However, function of one object can access the function of other objects.

Organization of data and function in OOP



Some of the features of object oriented programming are:

- Emphasis is on data rather than procedure.
- Programs are divided into what are known as objects.
- Data structures are designed such that they characterize the objects.
- Functions that operate on the data of an object are tied together in the data structure.
- Data is hidden and cannot be accessed by external function.
- Objects may communicate with each other through function.
- New data and functions can be easily added whenever necessary.
- Follows bottom up approach in program design.

Object-oriented programming is the most recent concept among programming paradigms and still means different things to different people.

Basic concepts of c++

There are few principle concepts that form the foundation of object-oriented programming:

1. **Object**
2. **Class**
3. **Data Abstraction & Encapsulation**
4. **Inheritance**
5. **Polymorphism**
6. **Dynamic Binding**

7. Message Passing

1) Object :

Object is the basic unit of object-oriented programming. Objects are identified by its unique name. An object represents a particular instance of a class. There can be more than one instance of an object. Each instance of an object can hold its own relevant data.

An Object is a collection of data members and associated member functions also known as methods.

For example whenever a class name is created according to the class an object should be created without creating object can't able to use class.

The class of Dog defines all possible dogs by listing the characteristics and behaviors they can have; the object Lassie is one particular dog, with particular versions of the characteristics. A Dog has fur; Lassie has brown-and-white fur.

2) Class:

Classes are data types based on which objects are created. Objects with similar properties and methods are grouped together to form a Class. Thus a Class represents a set of individual objects. Characteristics of an object are represented in a class as Properties. The actions that can be performed by objects become functions of the class and is referred to as Methods.

When you define a class, you define a blueprint for an object. This doesn't actually define any data, but it does define what the class name means, that is, what an object of the class will consist of and what operations can be performed on such an object.

For example consider we have a Class of Cars under which Santro Xing, Alto and WaganR represents individual Objects. In this context each Car Object will have its own, Model, Year of Manufacture, Colour, Top Speed, Engine Power etc., which form Properties of the Car class and the associated actions i.e., object functions like Start, Move, Stop form the Methods of Car Class. No memory is allocated when a class is created. Memory is allocated only when an object is created, i.e., when an instance of a class is created.

3) Data abstraction & Encapsulation :

Encapsulation is placing the data and its functions into a single unit. While working with procedural languages, it is not always clear which functions work on which variables but object-oriented programming provides you framework to place the data and the relevant functions together in the same object.

When using **Data Encapsulation**, data is not accessed directly, it is only accessible through the functions present inside the class.

Data Abstraction increases the power of programming language by creating user defined data types. Data Abstraction also represents the needed information in the program without presenting the details.

Abstraction refers to the act of representing essential features without including the background details or explanation between them.

For example, a class Car would be made up of an Engine, Gearbox, Steering objects, and many more components. To build the Car class, one does not need to know how the different components work internally, but only how to interface with them, i.e., send messages to them, receive messages from them, and perhaps make the different objects composing the class interact with each other.

4) Inheritance :

One of the most useful aspects of object-oriented programming is code reusability. As the name suggests Inheritance is the process of forming a new class from an existing class or base class.

The base class is also known as parent class or super class, the new class that is formed is called derived class.

Derived class is also known as a child class or sub class. Inheritance helps in reducing the overall code size of the program, which is an important concept in object-oriented programming.

This is a very important concept of object-oriented programming since this feature helps to reduce the code size.

It is classified into different types, they are

- **Single level inheritance**
- **Multi-level inheritance**
- **Hybrid inheritance**
- **Hierarchical inheritance**

5) Polymorphism :

Polymorphism allows routines to use variables of different types at different times. An operator or function can be given different meanings or functions. Polymorphism refers to a single function or multi-functioning operator performing in different ways. Poly a Greek term means the ability to take more than one form. Overloading is one type of Polymorphism. It allows an object to have different meanings, depending on its context. When an existing operator or function begins to operate on new data type, or class, it is understood to be overloaded.

6) Dynamic binding :

Binding means connecting one program to another program that is to be executed in reply to the call. Dynamic binding is also known as late binding. The code present in the specified program is unknown till it is executed. It contains a concept of Inheritance and Polymorphism.

7) Message Passing :

An object-oriented program consists of a set of objects that communicate with each other. The process of programming in an object-oriented language, therefore, involves the following basic steps:

1. Creating classes that define objects and their behaviour
2. Creating objects from class definitions and
3. Establishing communication among objects.

Objects communicate with one another by sending and receiving information much the same way as people pass messages to one another.

A message for an object is a request for execution of a procedure, and therefore will invoke a function in the receiving object that generates the desired result. Message passing involves specifying the name of the object, the name of the function and the information to be sent.

Using main() function, Compiling and Executing Simple Programs in C++

Simple C++ Program

Let us begin with a simple example of a C++ program that prints a string on the screen.

program 1.10.1

```
#include<iostream> Using namespace std; int main()
{
cout<<" c++ is better than c \n";
return 0;
}
```

1.10.1 Program feature

Like C, the C++ program is a collection of function. The above example contain only one function **main()**. As usual execution begins at **main()**. Every C++ program must have a **main()**. C++ is a free form language. With a few exception, the compiler ignore carriage return and white spaces. Like C, the C++ statements terminate with semicolons.

1.10.2 Comments

C++ introduces a new comment symbol // (double slash). Comment start with a double slash symbol and terminate at the end of the line. A comment may start anywhere in the line, and whatever follow still the end of the line is ignored. Note that there is no closing symbol.

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

```
// This is an example of
// C++ program to illustrate
// some of its features
```

The C comment symbols /*,*/ are still valid and are more suitable for multiline comments. The following comment is allowed:

```
/* This is an example of C++ program to illustrate some of its features
*/
```

1.10.3 Output operator

The only statement in program 1.10.1 is an output statement. The statement

```
Cout<<"C++ is better than C.";
```

Causes the string in quotation marks to be displayed on the screen. This statement introduces two new C++ features, `cout` and `<<`. The identifier `cout` (pronounced as Cout) is a predefined object that represents the standard output stream in C++. Here, the standard output stream represents the screen. It is also possible to redirect the output to other output devices. The operator `<<` is called the insertion or put to operator.

1.10.4 The iostream File

The following `#include` directive has been used in the program:

```
#include <iostream>
```

The `#include` directive instructs the compiler to include the contents of the file enclosed within angular brackets into the source file. The header file **`iostream.h`** should be included at the beginning of all programs that use input/output statements.

1.10.5 Namespace

Namespace is a new concept introduced by the ANSI C++ standards committee. This defines a scope for the identifiers that are used in a program. For using the identifier defined in the **namespace** scope we must include the `using` directive, like

```
Using namespace std;
```

Here, `std` is the namespace where ANSI C++ standard class libraries are defined. All ANSI C++ programs must include this directive. This will bring all the identifiers defined in `std` to the current global scope. **Using** and **namespace** are the new keywords of C++.

1.10.6 Return Type of main()

In C++, `main()` returns an integer value to the operating system. Therefore, every `main ()` in

C++ should end with a return (0) statement; otherwise a warning or an error might occur. Since main() returns an integer type for main() is explicitly specified as **int**. Note that the default return type for all function in C++ is **int**. The following main without type and return will run with a warning:

```
main ()
{
.....
.....
}
```

1.11 More C++ Statements

Let us consider a slightly more complex C++ program. Assume that we should like to read two numbers from the keyboard and display their average on the screen. C++ statements to accomplish this is shown in program 1.11.1

AVERAGE OF TWO NUMBERS

```
#include<iostream.h> // include header file
```

```
using namespace std; int main()
{
```

```
float number1, number2, sum, average; cin >> number1; // Read Numbers cin >>
number2; // from keyboard sum = number1 + number2;
Average = sum/2;
cout << "Sum = " << sum << "\n";
cout << "Average = " << average << "\n"; return 0;
} //end of example
```

The output would be:

```
Enter two numbers: 6.5 7.5
Sum= 14
Average = 7
```

1.11.1 Variable

The program uses four variables number1, number2, sum and average. They are declared as type float by the statement.

```
float number1, number2, sum, average;
```

All variable must be declared before they are used in the program.

1.11.2 Input Operator

The statement

```
cin>> number1;
```

is an input statement and causes the program to wait for the user to type in a number. The number keyed in is placed in the variable number 1. The identifier cin (pronounced 'C in') is a predefined object in C++ that corresponds to the standard input stream. Here, this stream represents the keyboard.

The operator >> is known as extraction or get from operator. It extracts (or takes) the value from the keyboard and assigns it to the variable on its right. This corresponds to a familiar scanf() operation. Like <<, the operator >> can also be overloaded.

1.11.3 Cascading of I/O Operators

We have used the insertion operator << repeatedly in the last two statements for printing results.

The statement

```
Cout<< "Sum= " << sum<< "\n";
```

First sends the string "Sum =" to cout and then sends the value of sum. Finally, it sends the new line character so that the next output will be in the new line. The multiple use of << in one statement is called cascading. When cascading an output operator, we should ensure necessary blank spaces between different items. Using the cascading technique, the last two statements can be combined as follows:

```
Cout<< "Sum= " << sum<< "\n"  
<< "Average = " << average << "\n";
```

This is one statement but provides two line of output. If you want only one line of output, the statement will be:

```
Cout<< "Sum= " << sum<< ", "  
<< "Average = " << average << "\n";
```

The output will be:

Sum= 14, average = 7

We can also cascade input operator >> as shown below: `Cin>> number1 >> number2;`
The values are assigned from left to right. That is, if we key in two values, say, 10 and 20, then 10 will be assigned to number1 and 20 to number2.

1.12 An Example with Class

One of the major features of C++ is classes. They provide a method of binding together data and functions which operate on them. Like structures in C, classes are user-defined data types.

Program 1.12.1 shows the use of class in a C++ program.

USE OF CLASS

```
#include <iostream.h> // include header file
using namespace std;
class person
{
    char name[30]; int age;
public:
    void getdata(void);
    void display(void);
};
void person :: getdata(void)
{
    cout<< "Enter name: ";
    cin>> name;
    cout<< "Enter age: ";
    cin>> age;
```

```
}  
Void person : : display(void)  
{  
cout<< "\nName: " << name;  
cout<< "\nAge: " << age;  
}  
  
int main()  
{  
person p; p.getdata(); p.display();  
Return 0;  
} //end of example
```

The output of program is:

```
Enter Name: Ravinder  
Enter age:30  
Name:Ravinder  
Age: 30
```

The program defines **person** as a new data type class. The class **person** includes two basic data type items and two functions to operate on that data. These functions are called **member functions**. The main program uses **person** to declare variables of its type. As pointed out earlier, class variables are known as objects. Here, **p** is an object of type **person**. Class objects are used to invoke the functions defined in that class.

1.13 Structure of C++ Program

As it can be seen from program 1.12.1, a typical C++ program would contain four sections as shown in fig. 1.9. This section may be placed in separate code files and then compiled independently or jointly.

It is a common practice to organize a program into three separate files. The class declarations are placed in a header file and the definitions of member functions go into another file. This approach enables the programmer to separate the abstract specification of the interface from the implementation details (member function definition).

Finally, the main program that uses the class is placed in a third file which "includes" the previous two files as well as any other file required.

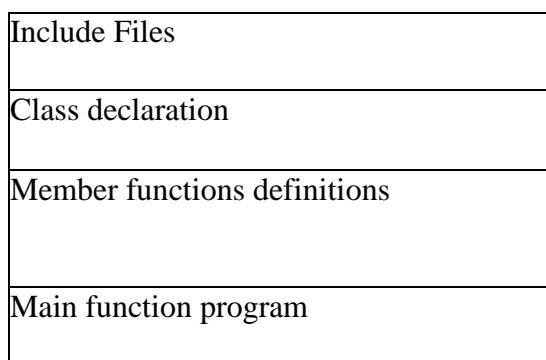
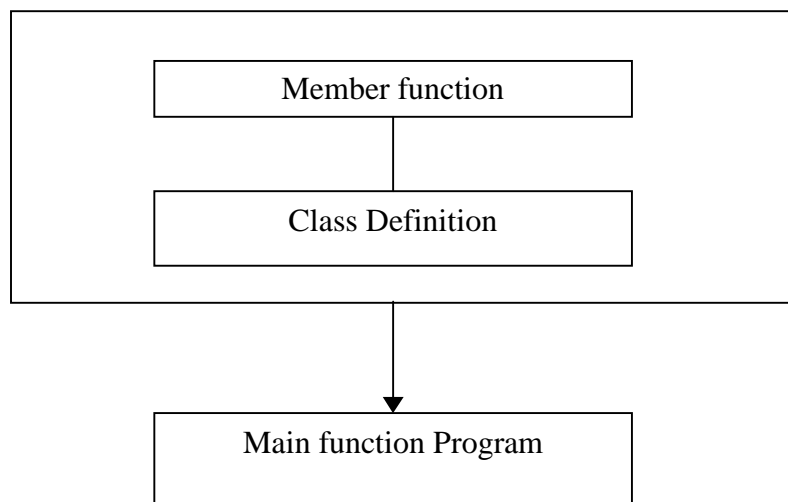


Fig 1.9 Structure of a C++ program

This approach is based on the concept of client-server model as shown in fig. 1.10. The class definition including the member functions constitute the server that provides services to the main program known as client. The client uses the server through the public interface of the class.

Fig. 1.10 The client-server model



1.14 Creating the Source File

Like C programs can be created using any text editor. For example, on the UNIX, we can use vi or ed text editor for creating and editing the source code. On the DOS system, we can use edlin or any other editor available or a word processor system under non-document mode.

Some systems such as TurboC, C++ provide an integrated environment for developing and editing programs.

The file name should have a proper file extension to indicate that it is a C++ implementation. Use extensions such as .c, .C, .cc, .cpp and .cxx. Turbo C++ and Borland C++ use .C for C programs and .cpp (Cplusplus) for C++ programs. Zortech C++ system uses .cxx while UNIX AT&T version uses .C (capital C) and .cc. The operating system manuals should be consulted to determine the proper file name extension to be used.

1.15 Compiling and Linking

The process of compiling and linking again depends upon the operating system. A few popular systems are discussed in this section.

Unix AT&T C++

This process of implementation of a C++ program under UNIX is similar to that of a C program. We should use the “cc” (uppercase) command to compile the program. Remember, we use lowercase “cc” for compiling C programs. The command

CC example.C

At the UNIX prompt would compile the C++ program source code contained in the file **example.C**. The compiler would produce an object file **example.o** and then automatically link with the library functions to produce an executable file. The default executable filename is **a.out**.

A program spread over multiple files can be compiled as follows:

CC file1.C file2.o

The statement compiles only the file **file1.C** and links it with the previously compiled **file2.o** file. This is useful when only one of the files needs to be modified. The files that are not modified need not be compiled again.

Data Types, Variables, Constants, Operators and Basic I/O:

Declaring, Defining and Initializing Variables, Scope of Variables

Declaration of variables

A variable provides us with named storage that our programs can manipulate. Each variable in C++ has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.

The name of a variable can be composed of letters, digits, and the underscore character. It must begin with either a letter or an underscore. Upper and lowercase letters are distinct because C++ is case-sensitive:

There are following basic types of variable in C++:

Type	Description
bool	Stores either value true or false.
char	Typically a single octet(one byte). This is an integer type.
Int	The most natural size of integer for the machine.
float	A single-precision floating point value.
double	A double-precision floating point value.
void	Represents the absence of type.
wchar_t	A wide character type.

C++ also allows defining various other types of variables which we will cover in subsequent chapters like **Enumeration, Pointer, Array, Reference, Data structures, and Classes**.

Following section will cover how to define, declare and use various type of variables.

Variable Declaration in C++:

All variables must be declared before use, although certain declarations can be made implicitly by content. A declaration specifies a type, and contains a list of one or more variables of that type as follows:

```
type variable_list;
```

Here, **type** must be a valid C++ data type including char, w_char, int, float, double, bool or any user defined object etc., and **variable_list** may consist of one or more identifier names separated by commas. Some valid declarations are shown here:

```
int i, j, k;
char c, ch;
float f, salary;
double d;
```

A variable declaration with an initializer is always a definition. This means that storage is allocated for the variable and could be declared as follows:

```
int i = 100;
```

An **extern** declaration is not a definition and does not allocate storage. In effect, it claims that a definition of the variable exists elsewhere in the program. A variable can be declared multiple times in a program, but it must be defined only once. Following is the declaration of a variable with extern keyword:

```
extern int i;
```

Though you can declare a variable multiple times in your C++ program but it can be declared only once in a file, a function or a block of code.

Variable Initialization in C++:

Variables are initialized (assigned an value) with an equal sign followed by a constant expression. The general form of initialization is:

```
variable_name = value;
```

Variables can be initialized (assigned an initial value) in their declaration. The initializer consists of an equal sign followed by a constant expression as follows:

```
type variable_name = value;
```

Some examples are:

```
int d = 3, f = 5;    // initializing d and f.
byte z = 22;        // initializes z.
double pi = 3.14159; // declares an approximation of pi.
char x = 'x';       // the variable x has the value 'x'.
```

For declarations without an initializer: variables with static storage duration are implicitly initialized with NULL (all bytes have the value 0); the initial value of all other variables is undefined.

It is a good programming practice to initialize variables properly otherwise, sometime program would produce unexpected result. Try following example which makes use of various types of variables:

```
#include <iostream>
using namespace std;
int main ()
{
    // Variable declaration:
    int a, b;
    int c;
    float f;
    // actual initialization
    a = 10;
    b = 20;
    c = a + b;
    cout<< c <<endl ;
    f = 70.0/3.0;
    cout<< f <<endl ;
    return 0;
}
```

When the above code is compiled and executed, it produces following result:

```
30
23.3333
```

Lvalues and Rvalues:

There are two kinds of expressions in C++:

1. **lvalue** : An expression that is an lvalue may appear as either the left-hand or right-hand side of an assignment.
2. **rvalue** : An expression that is an rvalue may appear on the right- but not left-hand side of an assignment.

Variables are lvalues and so may appear on the left-hand side of an assignment. Numeric literals are rvalues and so may not be assigned and can not appear on the left-hand side. Following is a valid statement:

```
int g = 20;
```

But following is not a valid statement and would generate compile-time error:

```
10 = 20;
```

Tokens

C++ Tokens are the smallest individual units of a program.

Following are the C++ tokens : (most of c++ tokens are basically similar to the C tokens)

- Keywords
- Identifiers
- Constants
- Variables
- Operators

Keywords (or reserved words):- These are the words used for special purposes or predefined tasks. These words should be written in small letters or lowercase letters. Some of the keyword used in c++ are int, float, char, double, long, void, for, while, do, if, else ...

There are 32 of these keywords, and they are:

```
autoconst double float int short struct unsigned
break continue else for long signed switch void
case default enumgoto register sizeoftypedef volatile
char do extern if return static union while
```

There are another 30 reserved words that were not in C, are therefore new to C++, and they are:

```
asmdynamic_cast namespace reinterpret_cast try
bool explicit new static_casttypeid
catch false operator template typename
class friend private this using
const_cast inline public throw virtual
delete mutable protected true wchar_t
```

The following 11 C++ reserved words are not essential when the standard ASCII character set is being used, but they have been added to provide more readable alternatives for some of the C++ operators, and also to facilitate programming with character sets that lack characters needed by C++.

andbitandcomplnot_eqor_eqxor_eq

and_eqbitor not or xor

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

There are some rules while defining identifiers:-

- i) The first character of identifier should be an alphabet or underscore. The rest can be letters or digits or underscore.
- ii) Special characters except underscore can't be part of identifiers.
- iii) Keywords can't be used as identifiers however keywords written in uppercase form can be used as identifiers.
- iv) Lower case and upper case letters are distinct.
- v) Maximum length of an identifier can be of 31 characters. However the length varies from one version of compiler to another version.

Some valid examples:- sum, a1, a2, _1, _a, average, a_b, x123y...

Some invalid examples:- 1a, a-b, float

Constants:- A value which remain constant throughout the program execution is known as constant. An identifier can be declared as a constant as illustrated below:-

i) # define N 20

ii) # define PI 3.14

iii) constint M=35; or const m=35;

Note:- When the constant identifier is of type integer, then keyword int is optional. As a default the identifier is considered as integer. Only one value can be declared as constant at a time with the keyword const.

Variables:- An identifier whose value can be modified/ altered is known as a variable. A variable can be declared as shown below.

int a;

float x;

charch;

Here a is a variable of type integer, x is a variable of type float, where asch is a variable of type char. Values can be assigned with a, x &ch as shown below.

a=10; x=7.5; ch='d'; a=a+5; x=x-4;

Literals:- The constant values used in c++ are known as literals, there are four type of literals namely.

i) **Integer constant:-** A value written without fraction part is known as integer constant. Example: 25, -674, 0 etc.

ii) **Floating constant:-** A value written with fraction part is floating value. Value of this type can be written with or without exponent form. Example: 2.34, -9.2154, 1.21E10

iii) **Character constant:-** A single character written within single quotation marks is known as character constant. Example: 'g', '9', '\$' etc

iv) **String constant:-** It is an array of characters enclosed in double quotation marks. Example: "Shubham", "03-aug-2009". Double quotation mark is a delimiter which determines length of a string.

Operators:- C++ is rich in supporting many types of operators as illustrated below.

A statement is a command given to the system to carry out a specific task. The statement consists of operators and operands where operators specify type of operation to be carried on operand.

Example:

```
c=a+b;
```

Here a and b are identifiers whose value is to be added and assigned in c. '+' and '=' are operators.

Operators in C++

Operator is a special symbol that tells the compiler to perform specific mathematical or logical Operation.

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Assignment Operators
- Ternary or Conditional Operators

	Operator	Type
unary operator →	++, --	Unary operator
Binary operator {	+, -, *, /, %	Arithmetic operator
	<, <=, >, >=, ==, !=	Relational operator
	&&, , !	Logical operator
	&, , <<, >>, ~, ^	Bitwise operator
	=, +=, -=, *=, /=, %=	Assignment operator
Ternary operator →	?: Tutorial4us.com	Ternary or conditional operator

Arithmetic Operators

Given table shows all the Arithmetic operator supported by C Language. Lets suppose variable **A** hold 8 and **B** hold 3.

Operator Example (int A=8, B=3) Result

+	A+B	11
-	A-B	5
*	A*B	24
/	A/B	2
%	A%4	0

Relational Operators

Which can be used to check the Condition, it always return true or false. Lets suppose variable **A** hold 8 and **B** hold 3.

Operators Example (int A=8, B=3) Result

<	A<B	False
<=	A<=10	True
>	A>B	True
>=	A<=B	False
==	A== B	False
!=	A!=(-4)	True

Logical Operator

Which can be used to combine more than one Condition?. Suppose you want to combined two conditions **A<B** and **B>C**, then you need to use **Logical Operator** like (A<B) && (B>C). Here && is Logical Operator.

Operator Example (int A=8, B=3, C=-10) Result

&&	(A<B) && (B>C)	False
	(B!=-C) (A==B)	True
!	!(B<=-A)	True

Truth table of Logical Operator

C1 C2 C1 && C2 C1 || C2 !C1 !C2

T	T	T	T	F	F
T	F	F	T	F	T
F	T	F	T	T	F
F	F	F	F	T	T

Assignment operators

Which can be used to assign a value to a variable.Lets suppose variable **A** hold 8 and **B** hold 3.

Operator Example (int A=8, B=3) Result

+=	A+=B or A=A+B	11
-=	A-=3 or A=A+3	5
=	A=7 or A=A*7	56
/=	A/=B or A=A/B	2
%=	A%=5 or A=A%5	3
=a=b	Value of b will be assigned to a	

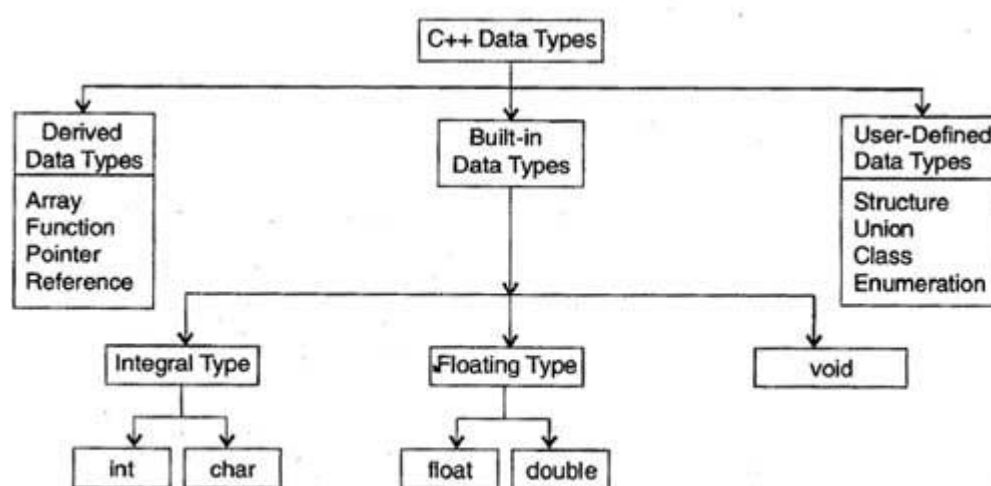
Left and Right Shift Operators

- The left shift operator (<<) is overloaded to designate stream output and is called **stream insertion operator**.
- The right shift operator (>>) is overloaded to designate stream input and is called **stream extraction operator**.
- These operators used with the standard stream object (and with other user defined stream objects) is listed below:

Operators	Brief description
<code>cin</code>	Object of istream class, connected to the standard input device , normally the keyboard.
<code>cout</code>	Object of ostream class, connected to standard output device , normally the display/screen.
<code>cerr</code>	Object of the ostream class connected to standard error device . This is unbuffered output, so each insertion to cerr causes its output to appear immediately.
<code>clog</code>	Same as cerr but outputs to clog are buffered.

Data Types in C++

A [data type](#) determines the type and the operations that can be performed on the data. C++ provides various data types and each data type is represented differently within the [computer's memory](#). The various data types provided by C++ are *built-in data types*, *derived data types* and *user-defined data types* as shown in Figure.



Various Data Types in C++

Built-In Data Types

The basic (fundamental) data types provided by c++ are *integral*, *floating point* and *void* data type. Among these data types, the integral and floating-point data types can be preceded by several typemodifiers. These modifiers (also known as type qualifiers) are the keywords that alter either size or range or both of the data types. The various modifiers are short, long, signed and unsigned. By default the modifier is signed.

In addition to these basic data types, ANSI C++ has introduced two more data types namely, `bool` and `wchar_t`.

Integral Data Type: The integral data type is used to store integers and includes `char` (character) and `int` (integer) data types.

Char: Characters refer to the alphabet, numbers and other characters (such as {, @, #, etc.) defined in the ASCII character set. In C++, the char data type is also treated as an integer data type as the characters are internally stored as integers that range in value from -128 to 127. The char data type occupies 1 byte of memory (that is, it holds only one character at a time).

The modifiers that can precede char are signed and unsigned. The various character data types with their size and range are listed in Table

Character Data Types		
Type	Size (in bytes)	Range
char	1	- 128 to 127
Signed char	1	- 128 to 127
unsigned char	1	0 to 255

Int: Numbers without the fractional part represent integer data. In C++, the int data type is used to store integers such as 4, 42, 5233, -32, -745. Thus, it cannot store numbers such as 4.28, -62.533. The various integer data types with their size and range are listed in Table

Integer Data Types		
Type	Size(in bytes)	Range
int	2	-32768 to 32767
signed int	2	-32768 to 32767
unsigned int	2	0 to 65535
short int	2	-32768 to 32767
signed short int	2	-32768 to 32767
unsigned short int	2	-32768 to 32767
long int	4	-2147483648 to 2147483647
signed long int	4	-2147483648 to 2147483647
unsigned long int	4	0 to 4294967295

Floating-point Data Type: A floating-point data type is used to store real numbers such as 3.28, 64.755765, 8.01, -24.53. This data type includes float and double' data types. The various floating -point data types with their size and range are listed in Table

Floating Point Data Types			
Type	Size(in bytes)	Range	Digits of Precision
float	4	$3.4 \cdot 10^{-38}$ to $3.4 \cdot 10^{38}$	7
double	8	$1.7 \cdot 10^{-308}$ to $1.7 \cdot 10^{308}$	15
long double	10	$3.4 \cdot 10^{-4932}$ to $3.4 \cdot 10^{4932}$	18

Void: The void data type is used for specifying an empty parameter list to a function and return type for a function. When void is used to specify an empty parameter list, it indicates that a function does not take any arguments and when it is used as a return type for a function, it indicates that a function does not return any value. For void, no memory is allocated and hence, it cannot store anything. As a result, void cannot be used to declare simple variables, however, it can be used to declare generic pointers.

Bool and wchar_t : The bool data type can hold only Boolean values, that is; either true or false, where true represents 1 and false represents 0. It requires only one bit of storage, however, it is stored as an integer in the memory. Thus, it is also considered as an integral data type. The bool data type is most commonly used for expressing the results of logical operations performed on the data. It is also used as a return type of a function indicating the success or the failure of the function.

In addition to char data type, C++ provides another data type wchar_t which is used to store 16-bit wide characters. Wide characters are used to hold large character sets associated with some non-English languages.

Derived Data Types: Data types that are derived from the built-in data types are known as derived data types. The various derived data types provided by C++ are *arrays*, *junctions*, *references* and *pointers*.

Array An array is a set of elements of the same data type that are referred to by the same name. All the elements in an array are stored at contiguous (one after another) memory locations and each element is accessed by a unique index or subscript value. The subscript value indicates the position of an element in an array.

Function A function is a self-contained program segment that carries out a specific well-defined task. In C++, every program contains one or more functions which can be invoked from other parts of a program, if required.

Reference A reference is an alternative name for a variable. That is, a reference is an alias for a variable in a program. A variable and its reference can be used interchangeably in a program as both refer to the same memory location. Hence, changes made to any of them (say, a variable) are reflected in the other (on a reference).

Pointer A pointer is a variable that can store the memory address of another variable. Pointers allow to use the memory dynamically. That is, with the help of pointers, memory can be allocated or de-allocated to the variables at run-time, thus, making a program more efficient.

User-Defined Data Types

Various user-defined data types provided by C++ are *structures*, *unions*, *enumerations* and *classes*.

Structure, Union and Class: Structure and union are the significant features of C language. Structure and union provide a way to group similar or dissimilar data types referred to by a single name. However, C++ has extended the concept of structure and union by incorporating some new features in these data types to support object-oriented programming.

C++ offers a new user-defined data type known as class, which forms the basis of object-oriented programming. A class acts as a template which defines the data and functions that are included in an object of a class. Classes are declared using the keyword `class`. Once a class has been declared, its object can be easily created.

Enumeration: An enumeration is a set of named integer constants that specify all the permissible values that can be assigned to enumeration variables. These set of permissible values are known as enumerators. For example, consider this statement.

```
enum country {US, UN, India, China};    // declaring an enum type
```

In this statement, an enumeration data-type `country` (`country` is a tag name), consisting of enumerators `US`, `UN` and so on, is declared. Note that these enumerators represent integer values, so any arithmetic operation can be performed on them.

By default, the first enumerator in the enumeration data type is assigned the value zero. The value of subsequent enumerators is one greater than the value of previous enumerator. Hence, the value of `US` is 0, value of `UN` is 1 and so on. However, these default integer values can be overridden by assigning values explicitly to the enumerators as shown here.

```
enum country {US, UN=3, India, china} ;
```

In this declaration, the value of `US` is 0 by default, the value of `UN` is 3, `India` is 4 and soon.

Once an enum type is declared, its variables can be declared using this statement.

```
country country1, country2;
```

These variables `country1`, `country2` can be assigned any of the values specified in enum declaration only. For example, consider these statements.

```
country1 India; // valid
```

```
country2 Japan; // invalid
```

Though the enumerations are treated as integers internally in C++, the compiler issues a warning, if an int value is assigned to an enum type. For example, consider these statements.

```
Country1 = 3;           //warning
```

```
Country1 = UN;         //valid
```

```
Country1 = (country) 3; //valid
```

C++ also allows creating special type of enums known as **anonymous enums**, that is, enums without using tag name as shown in this statement.

```
enum {US, UN=3, India, China};
```

The enumerators of an anonymous enum can be used directly in the program as shown here.

```
int count = US;
```

The typedef Keyword

C++ provides a typedef feature that allows to define new data type names for existing data types that may be built-in, derived or user-defined data types. Once the new name has been

defined, variables can be declared using this new name. For example, consider this declaration.

typedef int integer;

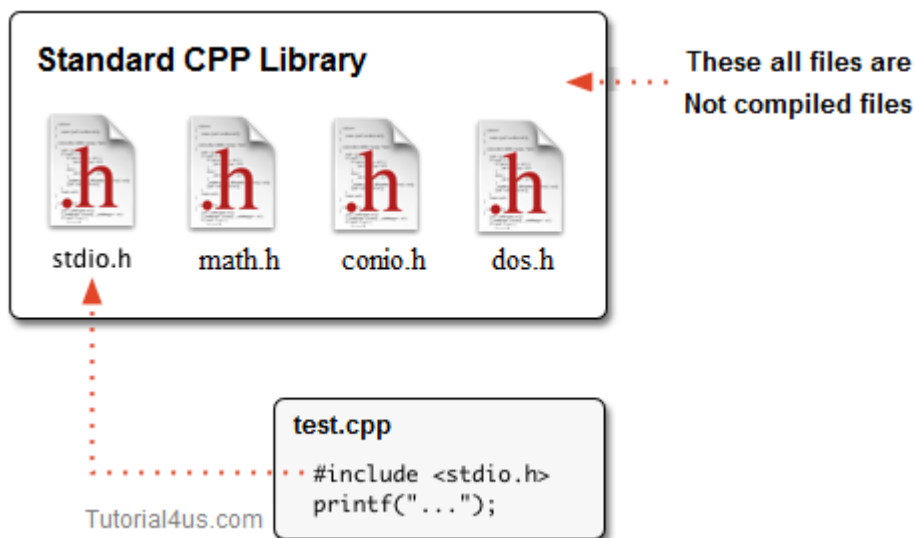
In this declaration, a new name **integer** is given to the data type **int**. This new name now can be used to declare integer variables as shown here.

integer i, j, k;

Note that the **typedef** is used in a program to contribute to the development of a clearer program. Moreover, it also helps in making machine-dependent programs more portable.

Header Files in C++

Header files contain definitions of **Functions and Variables**, which is imported or used into any C++ program by using the pre-processor **#include** statement. Header files have an extension **".h"** which contains C++ function declaration and macro definition.



Each header file contains information (or declarations) for a particular group of functions. Like **stdio.h** header file contains declarations of standard input and output functions available in C++ which is used for get the input and print the output. Similarly, the header file **math.h** contains declarations of mathematical functions available in C++.

Types of Header files

- **System header files:** It comes with compiler.
- **User header files:** It is written by programmer.

Why need of header files

When we want to use any function in our C++ program then first we need to import their definition from C++ library, for importing their declaration and definition we need to include header file in program by using `#include`. Header file include at the top of any C++ program.

For example if we use `clrscr()` in C++ program, then we need to include, `conio.h` header file, because in `conio.h` header file definition of `clrscr()` (for clear screen) is written in `conio.h` header file.

Syntax

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

See another simple example why use header files

Syntax

```
#include<iostream>
```

```
int main()
{
    using namespace std;
    cout<< "Hello, world!" <<endl;
    return 0;
}
```

In above program print message on screen hello world! by using `cout` but we don't define `cout` here actually already `cout` has been declared in a header file called **`iostream`**.

How to use header file in Program

Both user and system header files are include using the pre-processing directive `#include`. It has following two forms:

Syntax

```
#include<file>
```

This form is used for system header files. It searches for a file named file in a standard list of system directives.

Syntax

```
#include"file"
```

This form used for header files of our own program. It searches for a file named file in the directive containing the current file.

Note: The use of angle brackets <> informs the compiler to search the compilers include directory for the specified file. The use of the double quotes "" around the filename inform the compiler to search in the current directory for the specified file.

Most programs will use iostream, BUT there are many others that are also commonly used. The older C header file names are prefixed with the letter 'c'. Here are some of the most common.

Input / Output

#include <iostream> Stream I/O. cout and cin, istream and ostream, and endl, fixed, and showpoint manipulators.

#include <iomanip> More I/O manipulators: eg, setw(w) and setprecision(p).

#include <fstream> File I/O. ifstream and ofstream.

#include <sstream> I/O to and from strings. istringstream and ostringstream.

C functions

#include <cassert> assert macro.

#include <cstring> C-string (array of chars) functions (strcpy(), ...).

#include <cctype> char functions.

#include <cstdlib> abs(), exit(), ...

#include <climits> CHAR_BIT (bits per char), CHAR_MIN, CHAR_MAX, SHRT_MIN, SHRT_MAX, INT_MIN, INT_MAX, LONG_MIN, LONG_MAX, ... C++ defines a numeric_limits template.

#include <cmath> FLT_MIN, FLT_MAX, FLT_DIG, DBL_MIN, DBL_MAX, DBL_DIG, ... C++ defines a numeric_limits template.

STL (Standard Template Library)

#include <string> String type and functions.

#include <vector> Fast insertion/deletion at back, random access. Implementation: dynamic array allocation/reallocation.

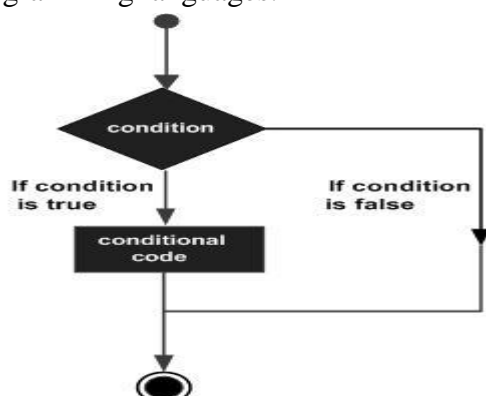
#include <list> Fast insertion/deletion everywhere. No direct access. Implementation: doubly linked list.

#include <deque>	Double-ended queue with fast insertion/deletion at front and back, direct access (one-level indirection). Implementation: multiple dynamically allocated blocks.
#include <stack>	stack - LIFO access, based on deque.
#include <queue>	queue (FIFO, based on deque) and priority_queue (retrieval of highest priority element first, based on vector).
#include <map>	map and multimap classes. Fast lookup based on key yields single/multiple entries. Implementation: balanced binary tree.
#include <set>	set and multiset containing single/multiple values. Fast lookup. Implementation: balanced binary tree.

Control Statements

Decision making structures require that the programmer specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

Following is the general form of a typical decision making structure found in most of the programming languages:



C++ programming language provides following types of decision making statements

Statement	Description
<u>if statement</u>	An if statement consists of a boolean expression followed by one or more statements.
<u>if...else statement</u>	An if statement can be followed by an optional else statement, which executes when the boolean expression is false.
<u>switch statement</u>	A switch statement allows a variable to be tested for equality against a list of values.
<u>nested if statements</u>	You can use one if or else if statement inside another if or else if statement(s).
<u>nested switch statements</u>	You can use one switch statement inside another switch statement(s).

The ? : Operator:

We have covered [conditional operator ?](#) : in previous chapter which can be used to replace **if...else** statements. It has the following general form:

```
Exp1?Exp2:Exp3;
```

Where Exp1, Exp2, and Exp3 are expressions. Notice the use and placement of the colon. The value of a ?expression is determined like this: Exp1 is evaluated. If it is true, then Exp2 is evaluated and becomes the value of the entire ?expression. If Exp1 is false, then Exp3 is evaluated and its value becomes the value of the expression.

If statement

An **if** statement consists of a boolean expression followed by one or more statements.

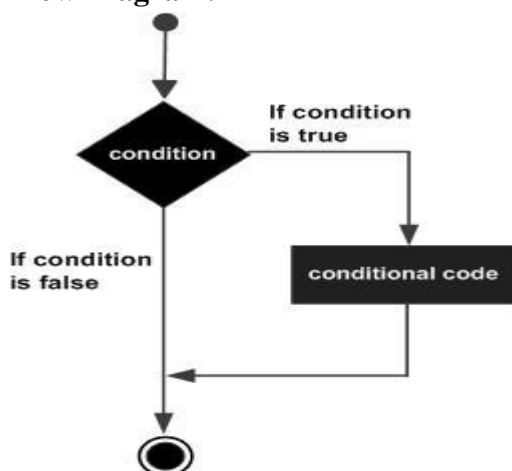
Syntax:

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

```
if(boolean_expression)
{
// statement(s) will execute if the boolean expression is true
}
```

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

Flow Diagram:



Example:

```
#include<iostream>
using namespace std;
int main ()
{
// local variable declaration:
int a =10;
// check the boolean condition
if( a <20)
{
// if condition is true then print the following
cout<<"a is less than 20;"<<endl;
```



```
}  
cout<<"value of a is : "<< a <<endl;  
return0;  
}
```

When the above code is compiled and executed, it produces the following result:

```
ais less than 20;  
value of a is:10
```

if-else statement

An **if** statement can be followed by an optional **else** statement, which executes when the boolean expression is false.

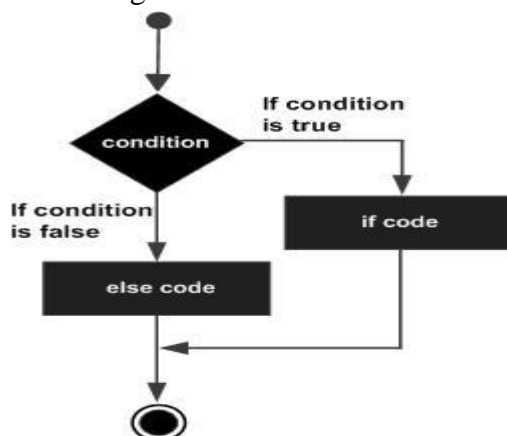
Syntax:

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

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

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

Flow Diagram:



Example:

```
#include<iostream>  
usingnamespacestd;  
int main ()  
{  
    // local variable declaration:  
    int a =100;
```

```
// check the boolean condition
if( a <20)
{
// if condition is true then print the following
cout<<"a is less than 20;"<<endl;
}
else
{
// if condition is false then print the following
cout<<"a is not less than 20;"<<endl;
}
cout<<"value of a is : "<< a <<endl;
return0;
}
```

When the above code is compiled and executed, it produces the following result:

```
aisnot less than 20;
value of a is:100
```

The if...else if...else Statement:

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

When using if , else if , else statements there are few points to keep in mind.

- An if can have zero or one else's and it must come after any else if's.
- An if can have zero to many else if's and they must come before the else.
- Once an else if succeeds, none of the remaining else if's or else's will be tested.

Syntax:

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

```
if(boolean_expression1)
{
// Executes when the boolean expression 1 is true
}
elseif(boolean_expression2)
{// Executes when the boolean expression 2 is true
}
elseif(boolean_expression3)
{// Executes when the boolean expression 3 is true
}
else
{// executes when the none of the above condition is true.
}
```

Example:

```
#include<iostream>
usingnamespacestd;
int main ()
{
    // local variable declaration:
    int a =100;
    // check the boolean condition
    if( a ==10)
    {
        // if condition is true then print the following
        cout<<"Value of a is 10"<<endl;
    }
    elseif( a ==20)
    {
        // if else if condition is true
        cout<<"Value of a is 20"<<endl;
    }
    elseif( a ==30)
    {
        // if else if condition is true
        cout<<"Value of a is 30"<<endl;
    }
    else
    {
        // if none of the conditions is true
        cout<<"Value of a is not matching"<<endl;
    }
    cout<<"Exact value of a is : "<< a <<endl;
    return0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Value of a isnot matching
Exact value of a is:100
```

Switch- case

A **switch** statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each case.

Syntax:

The syntax for a **switch** statement in C++ is as follows:

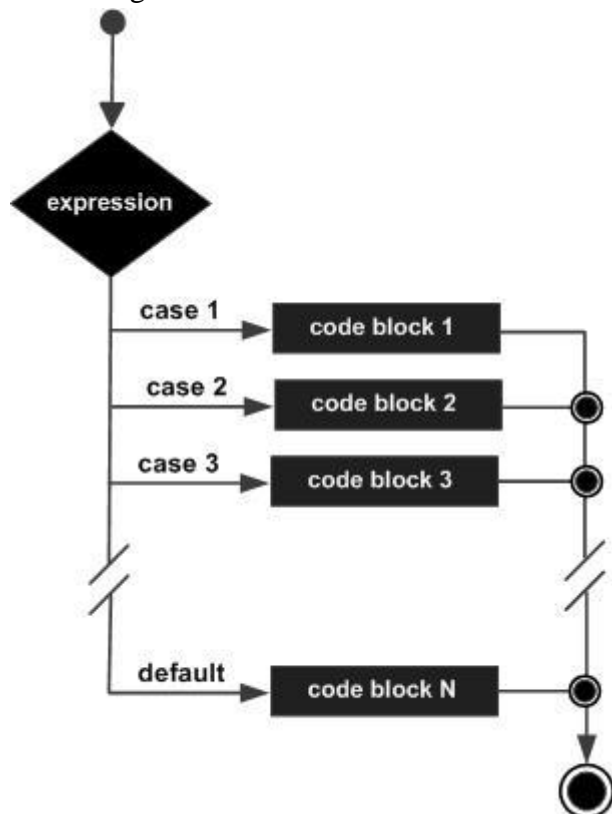
```
switch(expression){
case constant-expression :
statement(s);
break;//optional
case constant-expression :
statement(s);
break;//optional
```

```
// you can have any number of case statements.
default://Optional
statement(s);
}
```

The following rules apply to a switch statement:

- The **expression** used in a **switch** statement must have an integral or enumerated type, or be of a class type in which the class has a single conversion function to an integral or enumerated type.
- You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.
- The **constant-expression** for a case must be the same data type as the variable in the switch, and it must be a constant or a literal.
- When the variable being switched on is equal to a case, the statements following that case will execute until a **break** statement is reached.
- When a break statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.
- Not every case needs to contain a break. If no break appears, the flow of control will *fall through* to subsequent cases until a break is reached.
- A **switch** statement can have an optional **default** case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No break is needed in the default case.

Flow Diagram:



Example:

```
#include<iostream>
usingnamespacestd;
int main ()
{
// local variable declaration:
char grade ='D';
switch(grade)
{
case'A':
cout<<"Excellent!"<<endl;
break;
case'B':
case'C':
cout<<"Well done"<<endl;
break;
case'D':
cout<<"You passed"<<endl;
break;
case'F':
cout<<"Better try again"<<endl;
break;
default:
cout<<"Invalid grade"<<endl;
}
cout<<"Your grade is "<< grade <<endl;
return0;
}
```

This would produce the following result:

```
You passed
Your grade is D
```

LOOPS:

While loop

A **while** loop statement repeatedly executes a target statement as long as a given condition is true.

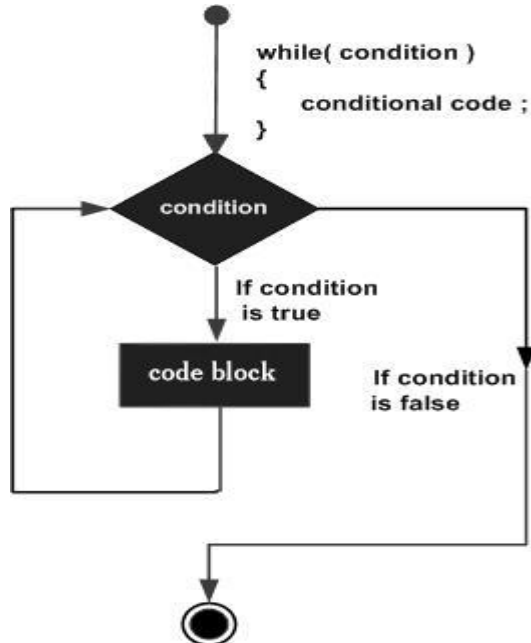
Syntax:

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

```
while(condition)
{
statement(s);
}
```

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

Flow Diagram:



Here, key point of the *while* loop is that the loop might not ever run. When the condition is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

Example:

```

#include<iostream>
usingnamespacestd;
int main ()
{
// Local variable declaration:
int a =10;
// while loop execution
while( a <20)
{
cout<<"value of a: "<< a <<endl;
a++;
}
return0;
}
  
```

When the above code is compiled and executed, it produces the following result:

```

value of a:10
value of a:11
value of a:12
  
```

```
value of a:13  
value of a:14  
value of a:15  
value of a:16  
value of a:17  
value of a:18  
value of a:19
```

for loop

A **for** loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

Syntax:

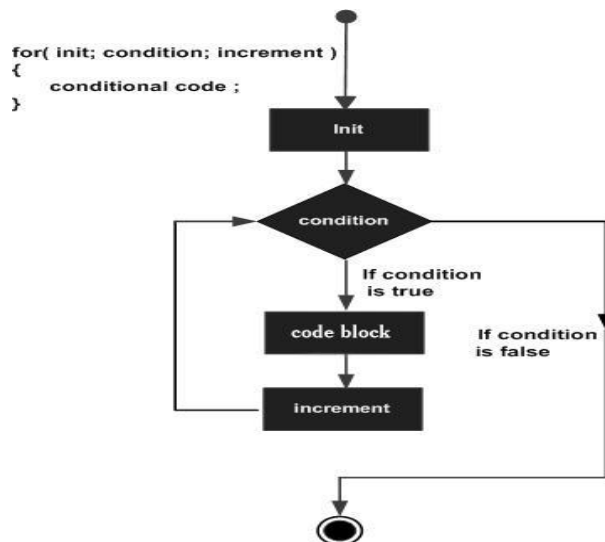
The syntax of a for loop in C++ is:

```
for(init; condition; increment )  
{  
statement(s);  
}
```

Here is the flow of control in a for loop:

- The **init** step is executed first, and only once. This step allows you to declare and initialize any loop control variables. You are not required to put a statement here, as long as a semicolon appears.
- Next, the **condition** is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and flow of control jumps to the next statement just after the for loop.
- After the body of the for loop executes, the flow of control jumps back up to the **increment** statement. This statement allows you to update any loop control variables. This statement can be left blank, as long as a semicolon appears after the condition.
- The condition is now evaluated again. If it is true, the loop executes and the process repeats itself (body of loop, then increment step, and then again condition). After the condition becomes false, the for loop terminates.

Flow Diagram:



Example:

```

#include<iostream>
usingnamespacestd;
int main ()
{
    // for loop execution
    for(int a =10; a <20; a = a +1)
    {
        cout<<"value of a: "<< a <<endl;
    }
    return0;
}

```

When the above code is compiled and executed, it produces the following result:

```

value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19

```

Do..while

Unlike **for** and **while** loops, which test the loop condition at the top of the loop, the **do...while** loop checks its condition at the bottom of the loop.

A **do...while** loop is similar to a while loop, except that a do...while loop is guaranteed to execute at least one time.

Syntax:

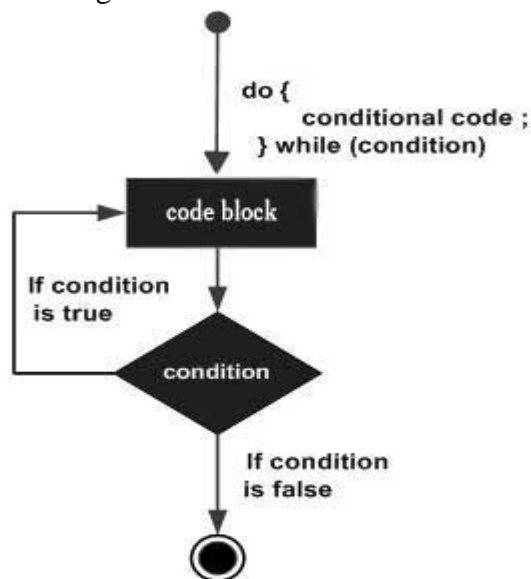
The syntax of a do...while loop in C++ is:

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

Notice that the conditional expression appears at the end of the loop, so the statement(s) in the loop execute once before the condition is tested.

If the condition is true, the flow of control jumps back up to do, and the statement(s) in the loop execute again. This process repeats until the given condition becomes false.

Flow Diagram:



Example:

```
#include<iostream>
usingnamespacestd;
int main ()
{ // Local variable declaration:
int a =10;
// do loop execution
do
{cout<<"value of a: "<< a <<endl;
a = a +1;
}while( a <20);
return0;
}
```

When the above code is compiled and executed, it produces the following result:

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
```

```

value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19

```

UNCONTIONAL JUMP STATEMENTS

C/C++ has four statements that perform an unconditional control transfer. These are `return()`, `goto`, `break` and `continue`. Of these `return()` is used only in functions. The `goto` and `return()` may be used anywhere in the program but `continue` and `break` statements may be used only in conjunction with a loop statement. In 'switch case' 'break' is used most frequently.

Go to Statement :

A **goto** statement provides an unconditional jump from the `goto` to a labeled statement in the same function.

NOTE: Use of **goto** statement is highly discouraged because it makes difficult to trace the control flow of a program, making the program hard to understand and hard to modify. Any program that uses a `goto` can be rewritten so that it doesn't need the `goto`.

Syntax:

The syntax of a `goto` statement in C++ is:

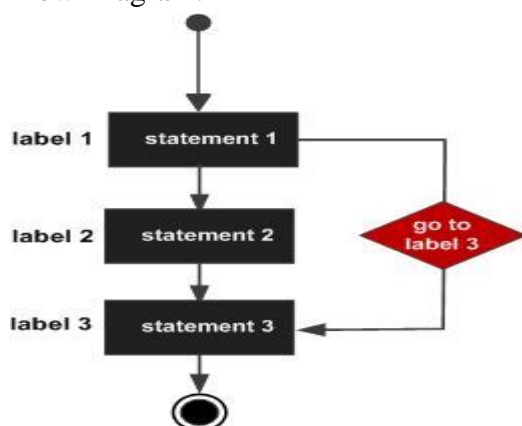
```

goto label;
..
label: statement;

```

Where **label** is an identifier that identifies a labeled statement. A labeled statement is any statement that is preceded by an identifier followed by a colon (:).

Flow Diagram:



Example:

```
#include<iostream.h>
usingnamespacestd;
int main ()
{
// Local variable declaration:
int a =10;

// do loop execution
LOOP:do
{if( a ==15)
{
// skip the iteration.
a = a +1;
goto LOOP;
}
cout<<"value of a: "<< a <<endl;
a = a +1;
}while( a <20);
return0;
}
```

When the above code is compiled and executed, it produces the following result:

```
value of a:10
value of a:11
value of a:12
value of a:13
value of a:14
value of a:16
value of a:17
value of a:18
value of a:19
```

One good use for the goto is to exit from a deeply nested routine. For example, consider the following code fragment:

```
for(...){
for(...){
while(...){
if(...)goto stop;
.
}
}
}
stop:
cout<<"Error in program.\n";
```

Eliminating the **goto** would force a number of additional tests to be performed.

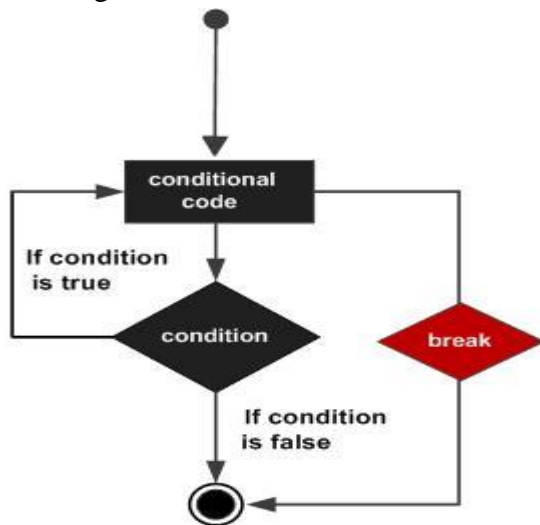
Break

Syntax:

The syntax of a break statement in C++ is:

```
break;
```

Flow Diagram:



Example:

```
#include<iostream>
usingnamespacestd;
int main ()
{
// Local variable declaration:
int a =10;
// do loop execution
do
{
cout<<"value of a: "<< a <<endl;
a = a +1;
if( a >15)
{
// terminate the loop
break;
}
}while( a <20);
return0;
}
```

When the above code is compiled and executed, it produces the following result:

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
```

The **break** statement has the following two usages in C++:

- When the **break** statement is encountered inside a loop, the loop is immediately terminated and program control resumes at the next statement following the loop.
- It can be used to terminate a case in the **switch**.

If you are using nested loops (i.e., one loop inside another loop), the break statement will stop the execution of the innermost loop and start executing the next line of code after the block.

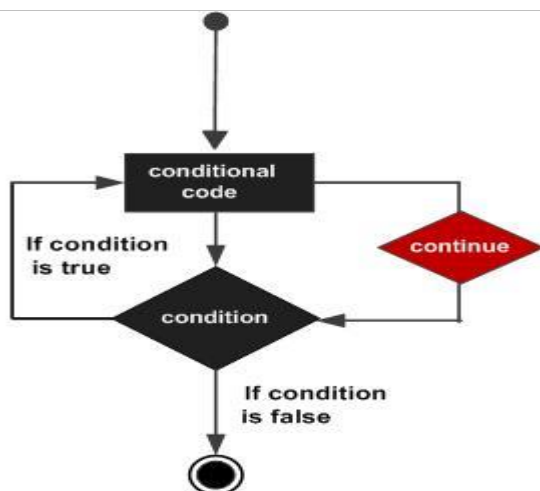
Continue

The **continue** statement works somewhat like the break statement. Instead of forcing termination, however, continue forces the next iteration of the loop to take place, skipping any code in between.

For the **for** loop, continue causes the conditional test and increment portions of the loop to execute. For the **while** and **do...while** loops, program control passes to the conditional tests.

The syntax of a continue statement in C++ is:

```
continue;
```



Example:

```
#include<iostream>
using namespace std;
int main ()
{
// Local variable declaration:
int a =10;
// do loop execution
```

```
do
{
if( a ==15)
{
// skip the iteration.
a = a +1;
continue;
}
cout<<"value of a: "<< a <<endl;
a = a +1;
}while( a <20);
return0;
}
```

When the above code is compiled and executed, it produces the following result:

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

KARPAGAM ACADEMY OF HIGHER EDUCATION

(Established Under Section 3 of UGC Act 1956)

COIMBATORE – 641 021

DEPARTMENT OF COMPUTER APPLICATIONS

Batch 2016 – 2019

16MMU304B Programming with C and C++

Part - B

(Each Question carries 2 marks)

1. What is the use of inheritance?
2. What is scope resolution operator? Give its syntax
3. Write the merits of encapsulation in OOPS.
4. Give some differences between variables and constants.
5. List the rules for naming an identifier in C++.
6. Write the general structure of a C++ program
7. What is Data Abstraction?
8. Define polymorphism.
9. Give the difference between else..if ladder and switch statements.
10. What is inline function? Give its syntax?
11. What do you mean by message passing?
12. List the advantages of OOPs.
13. Name the demerits of Structured programming over OOP.
14. Differentiate break and continue statements.
15. Discuss and analyze the merits of object oriented programming over structured programming with appropriate examples.
16. Describe the major parts of a c++ program and explain with an example program.
17. Describe the various control structures used in c++ with syntax and example.
18. Discuss in detail about the benefits of oops concept.
19. Write in detail about method overloading with an example program.
20. What is OOP? How is it different from procedure-oriented programming?
21. Give the evolution diagram of OOPS concept.
22. What is Procedure oriented language?
23. Give some characteristics of procedure-oriented language.
24. Write any four features of OOPS.
25. What are the basic concepts of OOS?

26. What are objects?
27. What is a class?
28. What is encapsulation?
29. What is data abstraction?
30. What are data members and member functions?
31. What is dynamic binding or late binding?
32. Write the process of programming in an object-oriented language?
33. List out the advantages of OOPS.
34. What are the features required for object-based programming Language?
35. What are tokens?

Part - C
(Each Question carries 6 marks)

1. Explain the concept of iterative statements in detail with a suitable example.
2. Illustrate a simple program to convert the Celsius into Fahrenheit.
3. Illustrate in detail the simple expression statements with suitable example program.
4. How to declare and initialize references? Explain with example.
5. Explain the conditional statements in detail with an appropriate example.
6. List out the characteristics of object oriented and procedure oriented programming.
7. Explain the basic concepts of object oriented programming in detail with an example.
8. Write a program to print the table using for loop.
9. Write a program to calculate the simple interest of a given values.
10. Explain the concept of operators in detail with a suitable example.
11. Describe the use of declaring and dereferencing pointers to simple variables.

KARPAGAM ACADEMY OF HIGHER EDUCATION

(Established Under Section 3 of UGC Act 1956)

COIMBATORE – 641 021

DEPARTMENT OF COMPUTER APPLICATIONS

Batch 2016 – 2019

Subject : Programming with C and C++

Subject Code: 16MMU304B

UNIT - I

S.No	Questions	opt1	opt2	opt3	opt4	Answer
1	The decomposition of a problem into a number of entities called_____	objects	classes	methods	messages	objects
2	OOPS follows_____ approach in program design	bottom-up	top-down	middle	top	bottom-up
3	Objects take up _____in the memory	space	address	memory	bytes	space
4	_____is a collection of objects of similar type	Objects	methods	classes	messages	classes
5	We can create _____of objects belonging to that class	1	2	10	any number	any number
6	The wrapping up of data & function into a single unit is known as _____	Polymorphism	encapsulation	functions	data members	encapsulation
7	_____refers to the act of representing essential features without including the background details	encapsulation	inheritance	Dynamic binding	Abstraction	Abstraction
8	Attributes are sometimes called_____	data members	methods	messages	functions	data members
9	The functions operate on the data are called_____	methods	data members	messages	classes	methods
10	_____is the process by which objects of one class acquire the properties of objects of another class	polymorphism	encapsulation	data binding	Inheritance	Inheritance
11	_____means the ability to take more than one form	polymorphism	encapsulation	data binding	Inheritance	polymorphism
12	The process of making an operator to exhibit different behaviors in different instances is known as _____	function overloading	operator overloading	method overloading	message overloading	operator overloading
13	Single function name can be used to handle different types of tasks is known as _____	function overloading	operator overloading	polymorphism	encapsulation	operator overloading
14	_____means that the code associated with a given procedure call is not known until the time of the call	late binding	Dynamic binding	Static binding	random binding	Dynamic binding
15	Objects can be_____	created	created & destroyed	permanent	temporary	created & destroyed
16	_____helps the programmer to build secure programs	Dynamic binding	Data hiding	Data building	message passing	Data hiding
17	_____techniques for communication between objects makes the interface descriptions with external systems much simpler	message passing	Data binding	Encapsulation	Data passing	message passing
18	Variables are declared in_____	only in main()	anywhere in the scope	before the main() only	only at the beginning	anywhere in the scope
19	How many sections in C++?	2	4	1	5	4
20	_____refers to permit initialization of the variables at run time	Dynamic initialization	Dynamic binding	Data binding	Dynamic message	Dynamic initialization
21	_____provides an alias for a previously defined variable	static variable	Dynamic variable	reference variable	address of an variable	reference variable
22	Reference variable must be initialized at the time of _____	declaration	assigning	initialization	running	declaration
23	The _____is an exit-controlled loop	while	do-while	for	switch	do-while
24	The _____is an entry-entrolled loop	while	do-while	for	switch	for
25	_____is an entry-controlled one	while	do-while	for	switch	while
26	Error checking does not occur during compilation if we are using_____	functions	macros	pre-defined functions	operators	macros
27	_____is a function that is expanded in line when it is invoked	macros	inline function	predefined function	preprocessor macros	inline function
28	_____refers to the use of same thing for different purposes	overloading	Dynamic binding	message loading	none	overloading
29	_____are extensively used for handling class objects	overloaded functions	methods	objects	messages	overloaded functions
30	_____is used to reduce the number of functions to be defined	default arguments	methods	objects	classes	default arguments
31	Control structures are said to be_____	programs	structured programs	statements	case statements	structured programs
32	_____is a decision making statement	for	jump	break	if	if
33	The bool type data occupies _____byte in memory	two	one	three	four	one

34	if-else-if ladder sometimes called_____	if-else-if nested	nested-if-else-if	if-else-if-staircase	if-else-if	if-else-if-staircase
35	How many statements are used to perform an unconditional transfer?	2	3	4	5	4
36	The label must start with_____	character	___	number	alphanumeric	character
37	_____statement is frequently used to terminate the loop in the switch case()	jump	goto	continue	break	break
38	_____statement does not require any condition	for	if	goto	while	goto
39	_____statement is used to transfer the control to pass on to the beginning of the block/loop	break	jump	goto	continue	continue
40	_____statement is a multiway branch statement	for	switch	if	while	switch
41	Every case statement in switch case statement terminates with	;	:	,	>>	:
42	How many types of loop control structure exist in c++?	1	3	2	4	3
43	The expression are separated by _____in the for loop	:	;	,	++	;
44	Test is performed at the _____of the for loop.	top	middle	end	program terminates	top
45	Condition is checked at the _____of the loop in the do-while statement.	beginning	end	middle	program terminates	end
46	Every expression always return_____	0 or 1	1 or 2	-1 or 0	-1 or 2	0 or 1
47	Which of the following loop statement uses 2 keyword?	do-while loop	for loop	if loop	while loop	do-while loop
48	The meaning of if(1) is_____	always false	always true	both(a) & (b)	flase	always true
49	The for loop comprises of _____actions	2	3	1	4	3
50	_____statement present at the bottom of the switch case statements	default	case	label	while	default
51	_____is an assignment statement that is used to set the loop control variables	Increment	declaring	Initialization	decrement	Initialization
52	Which of the following control expressions are valid for an of statement ?	an integer expression	a Boolean expression	either A or B	Neither A nor B	a Boolean expression
53	_____ is a subroutine that may include one or more statements designed to perform a specific task	structure	function	program	instruction	function
54	_____ section contains all the user defined functions that are called in the main function	link	documentation	definition	subprogram	subprogram
55	_____ is a group of related data items that share a common name	variables	array	function	structure	array
56	A _____ is an array of characters	string	variables	function	pointers	string
57	When the compiler assigns a character string to a character array it automatically supplies a _____ character at end of the string		null	dot	*	null
58	_____ function joins two strings together	strcat	strcmp	strcpy	strlen	strcat
59	_____ function compares two strings identified by the arguments	strcat	strcmp	strcpy	strlen	strcmp
60	strcmp function returns the value_____ if the arguments are equal	zero	one	two	three	zero

KARPAGAM ACADEMY OF HIGHER EDUCATION
(Established Under Section 3 of UGC Act 1956)
COIMBATORE – 641 021
DEPARTMENT OF COMPUTER APPLICATIONS

Subject : PROGRAMMING WITH C and C++

Sub.code : 16MMU304B

2.1 INTRODUCTION

Functions are the building blocks of C++ programs where all the program activity occurs. Function is a collection of declarations and statements.

Monoethic program (a large single list of instructions) becomes difficult to understand. For this reason functions are used. A function has a clearly defined objective (purpose) and a clearly defined interface with other functions in the program. Reduction in program size is another reason for using functions. The functions code is stored in only one place in memory, even though it may be executed as many times as a user needs.

The following program illustrates the use of a function :

```
//to display general message using function
#include<iostream.h>
include<conio.h>
void main()
{
void disp(); //function prototype clrscr(); //clears the screen disp(); //function call
getch(); //freeze the monitor
}
//function definition void disp()
{
cout<<"Welcome to the GJU of S&T\n";
cout<<"Programming is nothing but logic implementation";
}
```

2.2 FUNCTION DEFINITION AND DECLARATION

In C++, a function must be defined prior to its use in the program. The function definition contains the code for the function. The general syntax of a function definition in C++ is shown below:

```
Type name_of_the_function (argument list)
{
//body of the function
}
```

Here, the type specifies the type of the value to be returned by the function. It may be any valid C++ data type. When no type is given, then the compiler returns an integer value from the function.

Name_of_the_function is a valid C++ identifier (no reserved word allowed) defined by the user and it can be used by other functions for calling this function.

Argument list is a comma separated list of variables of a function through which the function may receive data or send data when called from other function. The following function illustrates the concept of function definition:

```
//function definition add()
void add()
{
int a,b,sum;
cout<<"Enter two integers"<<endl;
cin>>a>>b;
sum=a+b;
cout<<"\nThe sum of two numbers is "<<sum<<endl;
}
```

The above function add () can also be coded with the help of arguments of parameters as shown below:

```
//function definition add()
void add(int a, int b) //variable names are must in definition
{
int sum;
sum=a+b;
cout<<"\nThe sum of two numbers is "<<sum<<endl;
}
```

2.3 ARGUMENTS TO A FUNCTION

Arguments(s) of a function is (are) the data that the function receives when called/invoked from another function.

2.3.1 PASSING ARGUMENTS TO A FUNCTION

It is not always necessary for a function to have arguments or parameters. The functions add () and divide () did not contain any arguments. The following example illustrates the concept of passing arguments to function SUMFUN ():

```
// demonstration of passing arguments to a function
```

```

#include<iostream.h>
void main ()
{
float x,result; //local variables int N;
formal parameters
Semicolon here float SUMFUN(float x, int N); //function declaration
return type
.....
.....
result = SUMFUN(X,N); //function declaration
}

//function SUMFUN() definition
    No semicolon here float SUMFUN(float x,int N) //function declaration
{
.....
.....          Body of the function
.....
}
No semicolon here

```

2.3.2 DEFAULT ARGUMENTS

C++ allows a function to assign a parameter the default value in case no argument for that parameter is specified in the function call. For example.

```

// demonstrate default arguments function
#include<iostream.h>
int calc(int U)
{
If (U % 2 == 0)
return U+10; Else
return U+2
}

Void pattern (char M, int B=2)
{
for (int CNT=0;CNT<B; CNT++) cout<<calc(CNT) <<M; cout<<endl;
}

Void main ()
{ Pattern('*'); Pattern ('#',4); Pattern (;@,3);
}

```

2.3.3 CONSTANT ARGUMENTS

A C++ function may have constant arguments(s). These arguments(s) is/are treated as constant(s). These values cannot be modified by the function.

For making the arguments(s) constant to a function, we should use the keyword `const` as given below in the function prototype :

```
Void max(const float x, const float y, const float z);
```

Here, the qualifier `const` informs the compiler that the arguments(s) having `const` should not be modified by the function `max ()`. These are quite useful when call by reference method is used for passing arguments.

2.4 CALLING FUNCTIONS

In C++ programs, functions with arguments can be invoked by:

- (a) Value
- (b) Reference

Call by Value: - In this method the values of the actual parameters (appearing in the function call) are copied into the formal parameters (appearing in the function definition), i.e., the function creates its own copy of argument values and operates on them. The following program illustrates this concept :

```
//calculation of compound interest using a function
#include<iostream.h>
#include<conio.h>
#include<math.h> //for pow()function
Void main()
{
Float principal, rate, time; //local variables
Void calculate (float, float, float); //function prototype clrscr();
Cout<<"\nEnter the following values:\n"; Cout<<"\nPrincipal:";
Cin>>principal;
Cout<<"\nRate of interest:"; Cin>>rate;
Cout<<"\nTime period (in yeasers) :"; Cin>>time;
Calculate (principal, rate, time); //function call
Getch ();
}

//function definition calculate()
Void calculate (float p, float r, float t)
```

```
{
Float interest; //local variable Interest = p* (pow((1+r/100.0),t))-p; Cout<<"\nCompound
interest is : "<<interest;
}
```

Call by Reference: - A reference provides an alias – an alternate name – for the variable, i.e., the same variable's value can be used by two different names : the original name and the alias name.

In call by reference method, a reference to the actual arguments(s) in the calling program is passed (only variables). So the called function does not create its own copy of original value(s) but works with the original value(s) with different name. Any change in the original data in the called function gets reflected back to the calling function.

It is useful when you want to change the original variables in the calling function by the called function.

//Swapping of two numbers using function call by reference

```
#include<iostream.h>
#include<conio.h>
void main()
{
clrscr();
int num1,num2;
void swap (int &, int &); //function prototype cin>>num1>>num2;
cout<<"\nBefore swapping:\nNum1: "<<num1;
cout<<endl<<"num2: "<<num2;
swap(num1,num2); //function call cout<<"\n\nAfter swapping : \Num1: "<<num1;
cout<<endl<<"num2: "<<num2;
getch();
}

//function definition swap()
void swap (int & a, int & b)
{
int temp=a;
a=b;
b=temp;
}
```

2.5 INLINE FUNCTIONS

These are the functions designed to speed up program execution. An inline function is expanded (i.e. the function code is replaced when a call to the inline function is made) in the line where it is invoked. You are familiar with the fact that in case of normal functions, the compiler have to jump to another location for the execution of the function and then the control is

returned back to the instruction immediately after the function call statement. So execution time taken is more in case of normal functions. There is a memory penalty in the case of an inline function.

The system of inline function is as follows :

```
inline function_header
{
body of the function
}
```

For example,

```
//function definition min()
inline void min (int x, int y)
cout<< (x < Y? x : y);
}
```

```
Void main()
{
int num1, num2;
cout<<"\nEnter the two intergers\n";
cin>>num1>>num2;
min (num1,num2; //function code inserted here
-----
}
}
```

An inline function definition must be defined before being invoked as shown in the above example. Here min () being inline will not be called during execution, but its code would be inserted into main () as shown and then it would be compiled.

If the size of the inline function is large then heavy memory pentaly makes it not so useful and in that case normal function use is more useful.

The inlining does not work for the following situations :

1. For functions returning values and having a loop or a switch or a goto statement.
2. For functions that do not return value and having a return statement.

3. For functions having static variable(s).
4. If the inline functions are recursive (i.e. a function defined in terms of itself).

The benefits of inline functions are as follows :

1. Better than a macro.
2. Function call overheads are eliminated.
3. Program becomes more readable.
4. Program executes more efficiently.

2.6 SCOPE RULES OF FUNCTIONS AND VARIABLES

The scope of an identifier is that part of the C++ program in which it is accessible. Generally, users understand that the name of an identifier must be unique. It does not mean that a name can't be reused. We can reuse the name in a program provided that there is some scope by which it can be distinguished between different cases or instances.

In C++ there are four kinds of scope as given below :

1. Local Scope
2. Function Scope
3. File Scope
4. Class Scope

Local Scope:- A block in C++ is enclosed by a pair of curly braces i.e., '{' and '}'. The variables declared within the body of the block are called local variables and can be used only within the block. These come into existence when the control enters the block and get destroyed when the control leaves the closing brace. You should note the variable(s) is/are available to all the enclosed blocks within a block.

For example,

```
int x=100;
{ cout<<x<<endl; Int x=200;
{ cout<<x<<endl; int x=300;
{
cout<<x<<endl;
}
}
cout<<x<<endl;
}
```

Function Scope : It pertains to the labels declared in a function i.e., a label can be used inside the function in which it is declared. So we can use the same name labels in different functions.

For example,

```
//function definition add1()
void add1(int x,int y,int z)
{
int sum = 0; sum = x+y+z; cout<<sum;
}
//function definition add2()
void add2(float x,float y,float z)
{
Float sum = 0.0; sum = x+y+z; cout<<sum;
}
```

Here the labels x, y, z and sum in two different functions add1 () and add2 () are declared and used locally.

File Scope : If the declaration of an identifier appears outside all functions, it is available to all the functions in the program and its scope becomes file scope. For Example,

```
int x;
void square (int n)
{
cout<<n*n;
}
void main ()
{
int num;
..... cout<<x<<endl; cin>>num; square(num);
.....
}
```

Here the declarations of variable x and function square () are outside all the functions so these can be accessed from any place inside the program. Such variables/functions are called global.

Class Scope : In C++, every class maintains its own associated scope. The class members are said to have local scope within the class. If the name of a variable is reused by a class member, which already has a file scope, then the variable will be hidden inside the class. Member functions also have class scope.

2.7 DEFINITION AND DECLARATION OF A CLASS

A class in C++ combines related data and functions together. It makes a data type which is used for creating objects of this type.

Classes represent real world entities that have both data type properties (characteristics) and associated operations (behavior).

The syntax of a class definition is shown below :

```
Class name_of _class
{
private : variable declaration; // data member
Function declaration; // Member Function (Method)
protected: Variable declaration; Function declaration;
public : variable declaration;
Function declaration;
};
```

Here, the keyword class specifies that we are using a new data type and is followed by the class name.

The body of the class has two keywords namely :

(i) private

(ii) public

In C++, the keywords private and public are called access specifiers. The data hiding concept in C++ is achieved by using the keyword private. Private data and functions can only be accessed from within the class itself. Public data and functions are accessible outside the class also.

Data hiding not mean the security technique used for protecting computer databases. The security measure is used to protect unauthorized users from performing any operation (read/write or modify) on the data.

The data declared under Private section are hidden and safe from accidental manipulation. Though the user can use the private data but not by accident.

The functions that operate on the data are generally public so that they can be accessed from outside the class but this is not a rule that we must follow.

2.8 MEMBER FUNCTION DEFINITION

The class specification can be done in two part :

- (i) Class definition. It describes both data members and member functions.
 - (ii) Class method definitions. It describes how certain class member functions are coded.
- We have already seen the class definition syntax as well as an example. In C++, the member functions can be coded in two ways :

(a) Inside class definition

(b) Outside class definition using scope resolution operator (::)

The code of the function is same in both the cases, but the function header is different as explained below :

2.8.1 Inside Class Definition:

When a member function is defined inside a class, we do not require to place a membership label along with the function name. We use only small functions inside the class definition and such functions are known as inline functions.

In case of inline function the compiler inserts the code of the body of the function at the place where it is invoked (called) and in doing so the program execution is faster but memory penalty is there.

2.8.2 Outside Class Definition Using Scope Resolution Operator (::) :

In this case the function's full name (qualified_name) is written as shown:

Name_of_the_class :: function_name

The syntax for a member function definition outside the class definition is :

return_type name_of_the_class::function_name (argument list)

```
{  
body of function  
}
```

Here the operator::known as scope resolution operator helps in defining the member function outside the class. Earlier the scope resolution operator(::)was used in situations where a global variable exists with the same name as a local variable and it identifies the global variable.

2.9 DECLARATION OF OBJECTS AS INSTANCES OF A CLASS

The objects of a class are declared after the class definition. One must remember that a class definition does not define any objects of its type, but it defines the properties of a class. For utilizing the defined class, we need variables of the class type. For example,

```
Largest ob1,ob2; //object declaration
```

will create two objects ob1 and ob2 of largest class type. As mentioned earlier, in C++ the variables of a class are known as objects. These are declared like a simple variable i.e., like fundamental data types.

In C++, all the member functions of a class are created and stored when the class is defined and this memory space can be accessed by all the objects related to that class.

Memory space is allocated separately to each object for their data members. Member variables store different values for different objects of a class.

2.10 ACCESSING MEMBERS FROM OBJECT(S)

After defining a class and creating a class variable i.e., object we can access the data members and member functions of the class. Because the data members and member functions are parts of the class, we must access these using the variables we created. For functions are parts of the class, we must access these using the variable we created. For Example,

```
Class student
```

```
{
private:
char reg_no[10];
char name[30];
int age;
char address[25];
public :
void init_data()
{
- - - - //body of function
- - - -
}
void display_data()
}
};
student ob; //class variable (object) created
- - - -
- - - -
```

```
Ob.init_data(); //Access the member function ob.display_data(); //Access the member
function
```

```
-----
-----
```

Here, the data members can be accessed in the member functions as these have private scope, and the member functions can be accessed outside the class i.e., before or after the main() function.

2.11 STATIC CLASS MEMBERS

Data members and member functions of a class in C++, may be qualified as static. We can have static data members and static member function in a class.

2.11.1 Static Data Member: It is generally used to store value common to the whole class. The static data member differs from an ordinary data member in the following ways :

(i) Only a single copy of the static data member is used by all the objects. (ii) It can be used within the class but its lifetime is the whole program.

For making a data member static, we require :

(a) Declare it within the class. (b) Define it outside the class.

For example

Class student

```
{
Static int count; //declaration within class
-----
-----
-----
};
```

The static data member is defined outside the class as :

```
int student :: count; //definition outside class
```

The definition outside the class is a must.

We can also initialize the static data member at the time of its definition as:

```
int student :: count = 0;
```

If we define three objects as : student obj1, obj2, obj3;

2.11.2 Static Member Function: A static member function can access only the static members of a class. We can do so by putting the keyword static before the name of the function while declaring it for example,

```

Class student
{
Static int count;
----- public :
-----
-----
static void showcount (void) //static member function
{
Cout<<"count="<<count<<"\n";
}
};
int student ::count=0;

```

Here we have put the keyword static before the name of the function shwocount (). ways:

In C++, a static member function differs from the other member functions in the following

- (i) Only static members (functions or variables) of the same class can be accessed by a static member function.
- (ii) It is called by using the name of the class rather than an object as given below:

Name_of_the_class :: function_name

For example,

```
student::showcount();
```

2.12 FRIEND CLASSES

In C++ , a class can be made a friend to another class. For example, class TWO; // forward declaration of the class TWO class ONE

```

{
.....
.....
public:
.....
.....
friend class TWO; // class TWO declared as friend of class ONE
};

```

Now from class TWO , all the member of class ONE can be accessed.

Arrays:

Suppose that we want a program that can read in a list of numbers and sort that list, or find the largest value in that list. To be concrete about it, suppose we have 15 numbers to read in from a file and sort into ascending order. We could declare 15 variables to store the numbers, but then how could we use a loop to compare the variables to each other? The answer is that we cannot. The problem is that we would like to have variables with subscripts or something like them, so that we could write something like

```
max = 0;
for ( i = 0; i < 15 ; i++ ) {
if ( number_i > max )
max = number_i ;
}
```

where somehow the variable referred to by number_i would change as i changed.

You have seen something like this already with C++ strings: if we declare

```
string str;
then we can write a loop like
for ( int i = 0; i < str.size(); i++ )
cout << str[i] << " ";
```

in which each individual character in str is accessed using the subscript operator []. The characters in a string form what we call an array. An array is conceptually a linear collection of elements, indexed by subscripts, all of the same type. If we could create an array named number with 15 elements, it would look like this: number:

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

Each element could be accessed using the subscript operator, as in number[1] or number[7], and we could write a loop like

```
max = 0;
for ( i = 0; i < 15 ; i++ ) {
if ( number_i > max )
max = number[i] ;
}
```

This would make it possible to manipulate large collections of homogeneous data, meaning data of the same type, with a single subscripted variable. Such is possible in C and C++ and all modern programming languages. Arrays are of fundamental importance to algorithms and computer science.

Declaring a (one-Dimensional) Array

Syntax:

```
elementtype arrayname [ size_expression ]
```

where

- _ elementtype is any type that already exists
- _ arrayname is the name of the array
- _ size_expression is the number of elements in the array

This declares an array named arrayname whose elements will be of type elementtype , and that has size_expression many elements.

Examples

```
char fname[24]; // an array named fname with 24 chars
int grade[35]; // an array named grade with 35 ints
int pixel[1024*768]; // an array named pixel with 1024*768 ints
const int MAX_STUDENTS = 100;
double average[MAX_STUDENTS]; // an array named average with 100 doubles
string fruit[5]; // an array of 5 C++ strings
```

The element type of an array is often called its base type. The _rst example is an array with base type char, for example. One can say that fname is _an array of char._

Things to remember about arrays:

- _ The starting index of an array is 0, not 1.
- _ The last index is one less than the size of the array.
- _ If the array has size elements, the range is 0..size-1.
- _ Arrays contain data of a single type.
- _ An array is a sequence of consecutive elements in memory and the start of the array is the address of its first element.
- _ Because the starting address of the array in memory is the address of its _rst element, and all elements are the same size and type, the compiler can calculate the locations of the remaining elements. If B is the starting address of the array array, and each element is 4 bytes long, the elements are at addresses B, B + 4, B + 8, B + 12, and so on, and in general, element array[k] is at address B + 12k.
- _ Although C and C++ allow the size expression to be variable, you should not use a variable, for reasons having to do with concepts of dynamic allocation and the lifetime of variables.
- _ Use constants such as macro de_nitions or const ints as the sizes of arrays.

Initializing Declarations

Arrays can be initialized at declaration time in two different ways.

```
elementtype arrayname[size expr] = { list with <= sizeexpr vals };
elementtype arrayname[ ] = { list with any number of values };
```

Examples

```
#define MAXSCORES 200
#define NUMFRUIT 5
const int SIZE = 100;
double numbers[SIZE]; // not initialized
string fruit[NUMFRUIT] = {"apple","pear","peach","lemon","mango"};
int power[ ] = {0,1,2,4,9,16,25,36,49,64,81,100};
int counts[SIZE] = {0};
int score[MAXSCORES] = {1,1,1};
```

The first declaration declares but does not initialize the array named numbers. The next declares and initializes an array named fruit with five strings:

```
apple pear peach lemon mango
0      1      2      3      4
```

The third initializes an array named power, whose size is determined by the number of values in the brace-delimited list. When the array size is given in square brackets but the number of values in the list is less than the size, the remainder of the array is initialized to 0. In the fourth example, all elements will be set to 0, and in the last, the first three are set to 1 and the rest, to 0.

Rules

- _ If the array size is given in brackets, then the initialized list must have at most that many values in it. If it has fewer, the rest of the array is initialized to 0's.
- _ If the size is not given in brackets, then the size is equal to the number of elements in the initialize list.
- _ If there is no initializer list, none of the array elements are initialized. They are not set to 0.

Advice

- _ Always named constants for the sizes of arrays. It makes it easier to write, maintain, and read the code.

Accessing Array Elements

An element of an array is accessed by using the subscript operator . The syntax is:
arrayname [integer-valued-expression-within-range]

Examples

```
cout << fruit[0] ; // prints apple
```

```
cout << powers[1] << _ << powers[2] ; // prints 1 2
fruit[3] = _apricot_ ; // replaces _peach_ by _apricot_ in fruit[3]
counts[SIZE-10] = counts[SIZE-11] + 2; // sets counts[90] to counts[89] + 2 = 2
cout << score[power[4]]; // power[4] = 9, so this outputs scores[9]
```

Notice in the last two examples that the index expression can be arbitrary integer-valued expressions, even the value of an array of integers.

Loops and Arrays

Without loops it is very hard to use arrays. Conversely, with them they are easy to use. In general, a loop can be used to initialize an array, to modify all elements, to access all elements, or to search for elements within the array. An example follows.

Example 1. Finding a minimum element.

This example shows how an array can be initialized with values from an input stream using a for-loop. The for-loop guarantees that the array is not `_over_lled_` because it runs only as many times as the array has elements. The example uses the preceding declarations.

```
// read values into array from standard input
for ( i = 0 ; i < MAXSCORES; i++ )
    cin >> score [ i ] ;
// Let minscore be the first element as a starting guess
int minscore = score [ 0 ] ;
// Compare remaining array elements to current maxscore
for ( i = 1 ; i < MAXSCORES; i++ ) {
    if ( minscore > score [ i ] ) // if current element < minscore
        minscore = score [ i ] ; // make it new minscore
// At this point , maxscore >= score [ j ] , for all j = 0 , 1 , 2 , ... i
}
cout << "The minimum score is " << minscore << endl ;
```

In this example, the score array is filled from values entered on the standard input stream, cin. After the entire array has been filled, a variable named minscore is set to the value of score[0]. Then a second loop examines each element of score from score[1], to score[2], and so on up to score[MAXSCORE-1]. If an element is smaller than minscore, its value is copied into minscore. This implies that in each iteration of the second loop, minscore is the smallest of the elements seen so far. This is how one searches for the minimum value in an unsorted array.

Arrays in Functions

An array parameter in a function is written as a name followed by empty square brackets:
`result_type function_name (..., elementtype arrayname [] ,...) ;`

For example,

```
int max( int array[] );
void foo( int a[], int b[], double c[]);
```

The size of the array is not put between the square brackets. When you declare a function with an array parameter, you just put the type and a name for the parameter followed by a pair of empty square brackets. This tells the compiler that the corresponding argument will be an array of that type.

How does the function know how many elements are in the array?

It doesn't. Arrays do not know their size. It is not stored anywhere. The function acts on any array of any size whose type matches the base type.

So how can you write a function that works with an array parameter?

You must always pass the size of the array as an extra parameter. For example, if we want to write a function that finds the minimum in an integer array, the function prototype would be

```
int min( int array[], int size );
```

We would call the function by passing just the array name to it. not the name with square brackets after it.

For example

```
int score[MAXSCORES];
for ( i = 0; i < MAXSCORES; i++ )
    cin >> score[i];
cout << _The minimum score is _ << min(score, MAXSCORES);
```

Notice that the score array is passed to the min() function just by writing its name. The min() function

definition would be

```
int min ( int array [ ] , int size )
{
    int minvalue = array [ 0 ] ;
    // Compare remaining array elements to current minvalue
    for ( int i = 1 ; i < size ; i++ ) {
        if ( minvalue > array [ i ] )
```

```

minvalue = array [ i ] ;
// minvalue <= array [ j ] , f o r a l l j <= i
}
r e t u r n minvalue ;
}

```

When the function is called as in `cout << max(score, MAXSCORES)`, the `score` array is passed to the array parameter `array`, and within the function `array[i]` is in fact a reference to `score[i]`, not a copy of it, but a reference to it, i.e., another name for it. This means that changes made to `array[i]` within the function are actually made to `score[i]` outside of the function.

There are three types of parameter passing:

- _ call by value parameters
- _ call by reference parameters
- _ array parameters

Array parameters are like call by reference parameters _ changes made to the array are changes made to the corresponding array argument.

Multidimensional Arrays:

When the element type of an array is another array, it is said that the array is multidimensional:

```

// array of 2 arrays of 3 int each
int a[2][3] = { { 1, 2, 3 }, // can be viewed as a 2 × 3 matrix
               { 4, 5, 6 } }; // with row-major layout

```

Note that when array-to-pointer decay is applied, a multidimensional array is converted to a pointer to its first element (e.g., a pointer to its first row or to its first plane): array-to-pointer decay is applied only once.

```

int a[2];           // array of 2 int
int* p1 = a;        // a decays to a pointer to the first element of a

```

```

int b[2][3];        // array of 2 arrays of 3 int
// int** p2 = b;    // error: b does not decay to int**
int (*p2)[3] = b;    // b decays to a pointer to the first 3-element row of b

```

```

int c[2][3][4];     // array of 2 arrays of 3 arrays of 4 int
// int*** p3 = c;   // error: c does not decay to int***
int (*p3)[3][4] = c; // c decays to a pointer to the first 3 × 4-element plane of c

```

KARPAGAM ACADEMY OF HIGHER EDUCATION

(Established Under Section 3 of UGC Act 1956)

COIMBATORE – 641 021

DEPARTMENT OF COMPUTER APPLICATIONS

Batch 2016 – 2019

16MMU304B Programming with C and C++

Part - B

(Each Question carries 2 marks)

1. What is function prototype?
2. What are streams?
3. Write some of the properties of friend functions.
4. How to define a member function?
5. What is a pointer?
6. What is meant by free functions?
7. What is function prototype?
8. What is an inline function?
9. What is a default argument?
10. What are constant arguments?
11. How the class is specified?
12. How to create an object?
13. How to access a class member?
14. How the member functions are defined?
15. What is static data member?
16. What is static member function?
17. How the objects are used as function argument?

Part - C

(Each Question carries 6 marks)

1. Discuss on how to create and declare an one-dimensional array.
2. Describe the utility of functions. Write a simple program to explain the use of functions.
3. Elaborate the difference between declaration and definition of functions.
4. Write a simple program to illustrate the use of command line arguments in functions.
5. Define the terms virtual functions and pure virtual functions with a suitable program.
6. Describe in detail the need of overloading functions and operators with a suitable program.
7. Illustrate in detail the simple expression statements with suitable example program.
8. Elaborate the difference between declaration and definition of functions.
9. Write a simple program to illustrate the use of command line arguments in functions.
10. Explain about friend function with a suitable example program.
11. How to manipulate array elements using loops in c++?
12. Explain inline functions in detail with a suitable example program.

KARPAGAM ACADEMY OF HIGHER EDUCATION

(Established Under Section 3 of UGC Act 1956)

COIMBATORE – 641 021

DEPARTMENT OF COMPUTER APPLICATIONS

Batch 2016 – 2019

Subject : Programming with C and C++**Subject Code: 16MMU304B****UNIT - II**

S.No	Questions	opt1	opt2	opt3	opt4	Answer
1	C++ supports all the features of _____ as defined in C	structures	union	objects	classes	structures
2	A structure can have both variable and functions as _____	objects	classes	members	arguments	members
3	The class _____ describes the type and scope of its members	Functions	Class Declaration	objects	structures	Class Declaration
4	The class _____ describes how the class function are implemented	Function definition	declaration	arguments	class declaration	Function definition
5	The keywords private and public are known as _____	Labels	dynamic	visibility	const	Labels
6	The class members that have been declared as _____ can be accessed only from within the class	Private	public	static	protected	Private
7	The class members that have been declared as _____ can be accessed from outside the class also	Private	Public	static	protected	Public
8	The variables declare inside the class is known as----- -	Function variables	data members	member function	class declaration	data members
9	The functions which are declared inside the class are known as _____	Member function	member variables	data variables	class declaration	Member function
10	The class variables are known as _____	Functions	members	objects	class	objects
11	The scop resolution operator is _____	>>	::	<<	::	::
12	A _____ function can call another member function directly	Assignment	member	variables	greater than	member
13	A _____ member variable is initialized to zero when the first object of its class is created	Dynamic	constant	static	protected	static
14	_____ Variables are normally used to maintain values common to the entire class.	Private	protected	Public	static	static
15	When a copy of the entire object is passed to the function it is called as _____	Pass by reference	pass by function	pass by pointer	pass by value	pass by value
16	When the address of the object is transferred to the function it is called as _____	pass by reference	pass by function	pass by pointer	pass by value	pass by reference
17	A _____ function can be invoked like a normal function without the help of any object	Void	friend	inline	public	friend
18	The _____ member variables must be defined outside the class.	Static	private	public	protected	Static
19	A friend function, although not a member function, has full access right to the _____ members of the class	Static	private	public	protected	private
20	_____ enables an object to initialize itself when it is created	Destructor	constructor	overloading	polymorphism	constructor
21	_____ destroys the objects when they are no longer required	Destructor	constructor	overloading	polymorphism	Destructor
22	The _____ is special because its name is the same as the class name.	Destructor	static	constructor	polymorphism	constructor
23	A constructor that accepts no parameters is called the _____ constructor	Copy	default	multiple	polymorphism	default
24	Constructors are invoked automatically when the _____ are created	Datas	classes	objects	polymorphism	objects
25	Constructors cannot be _____	Inherited	destroyed	both a & b	polymorphism	Inherited
26	_____ is the collection of similar data type	classes	variable	both a & b	polymorphism	variable
27	Constructors make _____ calls to the operators new and delete when memory allocation is required	Explicit	implicit	function	polymorphism	implicit
28	The constructors that can take arguments are called _____ constructors	Copy	multiple	parameterized	polymorphism	parameterized
29	The constructor function can also be defined as _____ function	Friend	inline	default	polymorphism	inline

30	In ----- constructor the argument cannot be passed as a value.	Multiple	copy	default	polymorphism	copy
31	When more than one constructor function is defined in a class, then the constructor is said to be _____	Multiple	copy	default	overloaded	overloaded
32	The ----- constructor is also used to allocate memory while creating objects.	Explicit	implicit	dynamic	none of the above	dynamic
33	The ----- is invoked whenever an object of its associated class is created.	Default constructor	destructor	constructor	parameterized	constructor
34	The process of initializing through a copy constructor is known as _____ initialization	Overloaded	multiple	copy	none of the above	copy
35	When no ----- constructor is defined, the compiler supplies its own copy constructor.	Default	multiple	copy	dynamic	copy
36	Allocation of memory to objects at the time of their construction is known as _____ construction	Static	copy	dynamic	multiple	dynamic
37	We can create and use constant objects using _____ keyword before object declaration.	Static	new	const	int	const
38	A destructor is preceded by _____ symbol	Dot	asterisk	colon	tilde	tilde
39	A ----- is used to destroy the objects that have been created by a constructor	destructor	binding	class	copy constructor	destructor
40	C++ compiler has an ----- constructor which creates objects, even though it was not defined in class	implicit	dynamic	copy	explicit	implicit
41	Which is a valid method for accessing the first element of the array item?	item(1)	item[1]	item[0]	item(0)	item[0]
42	Which of the following statements is valid array declaration?	int number (5);	float avg[5];	double [5] marks;	counter int[5];	float avg[5];
43	An object is an _____ unit	group	individual	static	dynamic	individual
44	The scope operator is terminated by _____	Semicolon	comma	dot	colon	colon
45	A constructor that accepts no parameters is called the _____ constructor.	default	parameterized	implicit	dynamic	default
46	The memory for static data is allocated only _____	twice	thrice	once	several times	once
47	Static member functions can be invoked using _____ name	class	object	data	function	class
48	When a class is declared inside a function they are called as _____ classes.	global	invalid	local	multiple	local
49	_____ can be virtual	destructors	constructors	both a & b	multiple	destructor
50	The _____ doesn't have any argument	constructor	copy constructor	destructor	public	destructor
51	The _____ also allocates required memory.	constructor	destructor	both a & b	public	constructor
52	Any constructor or destructor created by the compiler will be _____	private	public	protected	global	public
53	The class can have only _____ destructor	two	many	one	four	one
54	_____ cannot be overloaded	destructor	constructor	friend	global	destructor
55	_____ releases memory space occupied by the objects	constructor	destructor	both a & b	friend	destructor
56	Constructors and destructors are automatically invoked by _____	operating system	main()	compiler	object	compiler
57	Constructors are executed when _____	object is destroyed	object is declared	when object is not used	object goes out of scope	object is declared
58	The destructor is executed when _____	object goes out of scope	when object is not used	when object contains nothing	object is destroyed	object goes out of scope
59	The members of a class are by default _____	protected	private	public	friend	private
60	The _____ is executed at the end of the function when objects are no longer used or go out of scope	destructor	constructor	inheritance	copy constructor	destructor
61	Which of the following cannot be passed to a function?	reference variables	arrays	class objects	header files	header files
62	Function should return a _____.	value	character	both (a) and (b)	integer	value
63	_____ function is useful when calling function is small	Built-in	Inline	user-defined	library	Inline
64	Inline function needs more _____	variables	functions	memory space	control structures	memory space
65	Multiple function with the same name is known as _____	function overloading	polymorphism	both a & b	operator overloading	function overloading
66	The _____ function creates a new set of variables and copies the values of arguments into them.	calling function	called function	both (a) and (b)	function	called function

67	Function contained within a class is called a _____	built-in	member function	user-defined function	calling function	member function
68	In c++,Declarations can appear_____in the body of the function	Only at the top	middle	bottom	anywhere	anywhere

KARPAGAM ACADEMY OF HIGHER EDUCATION

(Established Under Section 3 of UGC Act 1956)

COIMBATORE – 641 021

DEPARTMENT OF COMPUTER APPLICATIONS

Batch 2016-2019

Subject : PROGRAMMING WITH C and C++

Sub.code : 16MMU304B

Structures and Unions

Structures group members (data and functions) to create new data types. Structures encapsulate data members (usually different data types), much like functions encapsulate program statements. Unions are like structures, but data members overlay (share) memory, and unions may access members as different types. We use structures and unions in applications that need user-defined types, such as databases, windows, and graphics.

Structures

Structure definitions have several formats, all of which include data members and member functions. To show the formats for structure definitions, let's look at structure data members first. We'll examine structure member functions in the next section.

The first structure format appears frequently in header files.

```
struct struct_name {  
    Type data_member1;  
    Type data_member2;  
    Type data_memberN;  
};
```

The word `struct_name` is a new data type. Data members may be any type, including pointers, references, arrays, and other structures whose types do not have the same name as `struct_name`. The compiler does not allocate memory with this format because we define only a type and not a variable.

NOTE

Structure data members cannot have the same type name as their enclosing structure, but we can define pointers to the same structure type.

```
struct node {    // structure type  
    int data;    // data member  
    node *fwd;   // pointer to struct type  
    node *back;  // pointer to struct type  
};
```

Note how soon we drop the word `struct` from subsequent `node` definitions once we define it as a type.

The second structure format builds a structure from an existing structure type and includes an initialization list to initialize it.

```
struct struct_name name = { init_list };
```

The keyword `struct` is optional when you define `struct_name` elsewhere. This format allocates memory for structure `name`, which is type `struct_name`. The brace-enclosed initialization list is optional.

The third structure format combines the first and second formats.

```
struct struct_name {  
    Type data_member1;  
    Type data_member2;  
    Type data_memberN;  
} name = { init_list };
```

This format allocates memory, and the keyword `struct` is necessary to define a structure type. You may omit `struct_name` in this format if you don't need to use it later on in another structure definition.

NOTE

Structure initializations require braces with `init_list`. Consider the following.

```
struct X {    // structure definition  
    int num;  // data member  
};  
  
X a = 12;    // illegal - no conversion  
X a = { 12 }; // legal - initializes data member
```

The first initialization without braces generates a compilation error because the compiler attempts to convert an integer 12 to a structure X type. The second initialization with braces is correct. Always use pairs of braces to properly initialize your structures.

Member Functions

C++ also allows function declarations (called member functions) inside structures.

```
struct struct_name {  
    Type member_function1(signature);  
    Type member_function2(signature);  
};
```

Typically, member function names are distinct, but the name of a member function may be the same as another if their signatures differ. There is no size overhead for member functions inside structures (try `sizeof()` to convince yourself).

Member functions are essential in object-oriented programming because new data types combine functionality (member functions) with data (data members), all in a single unit. This concept lets you design objects with an implementation (how objects work) and an interface

(how objects behave). We explore these important concepts in Chapter 4 with class definitions.

The dot (.) operator provides access to structure members.

```
struct X {
    int num;        // data member
    int f();        // member function
};

X a = { 12 };      // initialize data member
cout << a.num << endl;    // data member, displays 12
cout << a.f() << endl;    // calls member function
```

Structure Pointers

Pointers may address structures. The formats are

```
struct struct_name {
    Type data_member;
    Type member_function(signature);
} *pname = { init_list };

struct struct_name *pname = { init_list };
```

The brace-enclosed `init_list` is optional and must contain a pointer expression whose type matches or converts to `struct_name`. If you initialize structure pointers, the braces surrounding `init_list` are not necessary. The word `struct_name` is optional in the first format, and the keyword `struct` is optional in the second format when you define `struct_name` elsewhere.

Here are the two formats to access structure members.

```
struct_name_variable.member_name // structure
struct_name_pointer->member_name // structure pointer
```

A structure variable name must appear to the left of the . (dot) operator and a structure pointer name must appear to the left of the -> (arrow) operator. Here are some examples of these operators with structure type block.

```
struct block {          // structure type
    int buf[80];        // data member
    char *pheap;        // data member
    void header(const char *); // member function
};

block data = { {1,2,3}, "basic" }; // structure variable
block *ps = &data;                // structure pointer

data.pheap++;                     // increment data member
data.header("magic");              // call member function
ps->pheap++;                       // increment data member
ps->header("magic");               // call member function
ps.pheap++;                      // illegal, ps is a pointer
data->header("magic");             // illegal, data is a structure
```

The inner braces in the initialization list for data initialize only the first three integers of the buf data member array (the remaining elements are zero). The precedence of . and -> is high; hence, no parentheses are necessary to combine them with a ++ operator when we increment pheap.

Be aware that memory allocation for structures does maintain the order of its data members in structure definitions. Structures, therefore, may contain "holes" due to machine word alignment (buf[79] in place of buf[80], for example).

Arrays with Structures

Arrays of structures and arrays of structure pointers are also possible. Two groups of formats exist. Here is the first one.

```
struct struct_name name[size] = { init_list };
struct struct_name name[size1][size2][sizeN] = { init_list };

struct struct_name *pname[size] = { init_list };
struct struct_name *pname[size1][size2][sizeN] = { init_list };
```

The keyword struct is optional when you define struct_name elsewhere, and the rules for size are the same as the rules for arrays of built-in types (see "Arrays" on page 35). The optional brace-enclosed init_list initializes a data member in each array element. With arrays of structure pointers (pname), init_list must have pointer expressions that match or convert to struct_name. When you initialize arrays of structures or arrays of structure pointers, you must include braces with init_list.

The second group of formats define a structure type and follow the same rules as above. The keyword struct must appear in both formats, but struct_name is optional.

```
struct struct_name {
    Type data_member;
    Type member_function(signature);
} name[size1][size2][sizeN] = { init_list };

struct struct_name {
    Type data_member;
    Type member_function(signature);
} *pname[size1][size2][sizeN] = { init_list };
```

Here are several examples of structure arrays.

```
struct block {          // structure type
    int buf[80];        // data member
    char *pheap;        // data member
    void header(const char *); // member function
};

block dbase[2] = {      // array of 2 structures
    { {1,2,3}, "one" }, // initialize first element
    { {4,5,6}, "two" }  // initialize second element
};
block *pb[2];          // array of 2 pointers to block
```

```

dbase[1].pheap++;    // increment data member
dbase[1].buf[0] = 'x';    // assign to data member
dbase[1].header("magic");    // call member function

pb[1] = &dbase[1];    // pb[1] points to dbase[1]
pb[1]->pheap++;    // increment data member
pb[1]->buf[0] = 'x';    // assign to data member
pb[1]->header("magic");    // call member function

```

NOTE

In the above example, we use an outer pair of braces on separate lines to initialize each member of the dbase structure array. Braces surrounding individual structure array elements make their initializations readable and easy to modify. Remember to use at least one pair of braces.

Nested Structures

Structure definitions that appear inside other structure definitions are nested structures. Here is an example.

```

struct team {
    struct address {    // nested structure
        char location[80];    // team location
        int zipcode;    // team zip code
    };
    char name[20];    // name of team
    address sponsor;    // sponsor's address
    address home_field;    // home field address
};

```

Nested structures encapsulate structure definitions and make them an integral part of an enclosing structure definition. Use the . or -> operators in succession to access the member you want from a nested structure.

```

team soccer = { "bears", {"123 Main", 30302}, {"8 Elm", 32240} };
team *ps = &soccer;    // structure pointer

cout << soccer.name << endl;    // displays "bears"
cout << soccer.sponsor.zipcode << endl;    // displays 30302
cout << ps->home_field.zipcode << endl;    // displays 32240

```

The first cout statement uses a . operator to display the team's name. Two . operators are necessary to display the zip code of the team's sponsor in the second cout statement. The third cout statement uses -> with structure pointer ps and . with structure data member home_field to display the team's home field zip code.

NOTE

Nested structures can use the same name as their enclosing structure names if they nest, too. Consider the following.

```

struct X {    // outer X
    struct Y {    // Y nested in outer X

```

```

struct X {          // inner X nested in Y
    int i;          // member of inner X
};
X mystuff;          // member of Y
int j;              // member of Y
};
Y stuff;            // member of outer X
int k;              // member of outer X
};

```

A structure Y nests inside outer structure X. Structure Y also nests an inner structure X, which is legal because its name is distinct from its immediately enclosing structure name (Y). Sometimes name collisions do occur when you piece together existing structures to form new ones, so be aware of this rule.

Typedefs with Structures

Chapter 2 introduces typedefs as synonyms for built-in types (see "Typedefs" on page 43). Typedefs also apply to structures and help make structure definitions more readable. Consider the structure definitions from the previous section.

```

struct node {
    int data;
    node *fwd;
    node *back;
};
node element;
node *p = &element;

```

Here's how we can simplify this code with the following typedef.

```

typedef struct node *Pnode;

struct node {
    int data;
    Pnode fwd;
    Pnode back;
};
node element;
Pnode p = &element;

```

Pnode is a synonym for struct node *. This typedef eliminates the pointer notation from our original code and makes it more readable.

NOTE

Create typedef names with uppercase first letters for better readability. Place typedef statements and structure definitions in header (.h) files and #include them where necessary. Remember that typedefs are not new types; they are just synonyms for existing ones. Typedefs, therefore, are not type safe.

Structure Copy and Assignment

Sometimes you want to save a structure temporarily or initialize a new structure from another one of the same type. The following statements attempt to copy the data members of one structure into another structure of the same type.

```
struct value {      // structure definition
    double x;
    char name[10];
} a = { 5.6, "start" }; // initialize members of struct a

value b;           // structure b of same type
b.x = a.x          // legal - copy doubles
b.name = a.name;   // illegal - not lvalues
```

Separate data member assignments are not only tedious for structures with a large number of members, but assignments fail with array members (array names are not lvalues). Fortunately, there is an easier way.

```
value b = a;       // structure copy
value c;
c = a;             // structure assignment
```

With structures of the same type, you may copy an existing structure to a new one or assign one structure to another. The compiler copies each member of one structure into the other (even array members!).

Remember that references to structures do not perform structure copies.

```
value f;           // structure
value & e = f;     // reference to a structure
```

The reference e is only an alias for structure f.

NOTE

Use structure copy and assignment statements. For built-in types (char, int, float, etc.), most compilers generate in assembly code block move instructions, which move data in memory without loops or counters. Thus, structure copy and assignment are efficient and convenient.

Unions

Unions have the same formats and operators as structures. Unlike structures, which reserve separate chunks of memory for each data member, unions allocate only enough memory to accommodate the largest data member. On 16-bit and 32-bit machines, for example, the definition

```
union jack {
    long number;
    char chbuf[4];
} chunk;
```


allocates only four bytes of memory. Variable `chunk.number` is a long, and `chunk.chbuf[2]` (for example) is a char, both within the same memory area. [Figure 3.3](#) shows the memory layout for `chunk`.



Figure 3.3. Memory layout of a union

You may create pointers to unions and initialize unions with an lvalue of the same data type as the union's first data member. Member functions are legal inside union definitions, but data members with constructors are not (see "Constructors" on page 181). Union assignment works just like structure assignment. Maintaining data integrity for union members is the programmer's responsibility.

C++ also has anonymous unions. An anonymous union allocates memory for its largest data member but doesn't create a type name (hence the name "anonymous"). Here's the format.

```
union {
    Type data_member1;
    Type data_member2;
    Type data_memberN;
};
```

A program that declares an anonymous union may access data members directly (without a `.` or `->`). Member functions are illegal inside anonymous union definitions.

To demonstrate anonymous unions, Listing 3.7 employs another version of `itoh()`, which converts short variables to hexadecimal characters for display.

Listing 3.7 Integer-to-hexadecimal conversion, using anonymous union

```
void itoh(unsigned short n) {
    union {                // anonymous union
        unsigned short num;
        unsigned char s[sizeof(short)];
    };
    const char *hex = "0123456789abcdef";
    num = n;                // store number as a short

    cout << hex[s[1] >> 4]; // first byte - high nibble
    cout << hex[s[1] & 15]; // first byte - low nibble

    cout << hex[s[0] >> 4]; // second byte - high nibble
    cout << hex[s[0] & 15]; // second byte - low nibble
}
```

An anonymous union creates two bytes (16 bits) of memory for `itoh()` to access. The statement `num = n` stores the number we pass to `itoh()` into memory as a short, and the `cout` statements access the same bytes of memory as characters. We use a 4-bit right shift (`>> 4`) to access the high nibble (4 bits) and a 4-bit mask (`& 15`) to access the low nibble. Both operations yield a 4-bit result between 0 and 15, which we use as an index with pointer `hex` to convert the result to ASCII characters ('0' to 'f'). The memory that our anonymous union creates is available only inside `itoh()`.

Why use this technique? First, this version of `itoh()` executes fast. Unlike other versions, there are no recursive calls, loops, or multiply or divide operators. Second, on some machines, the

assembly code from this function may be smaller in size than other versions. In time-critical code or with programs that have memory constraints, this version of `itoh()` may be preferable to others.

NOTE

Unions are not always portable, due to the way different processors store bytes in memory. Our version of `itoh()`, for instance, runs correctly on Intel \AA processors, but displays the bytes in reverse order on SPARC \AA -based workstations. The following preprocessor statements declare and initialize a `const` integer that identifies the machine we are using.

```
#ifndef SPARC
const int byte = 0;    // SPARC-based
#else
const int byte = 1;    // Intel
#endif
```

Inside `itoh()`, we modify the `cout` statements as follows.

```
cout << hex[s[byte] >> 4]; // first byte - high nibble
cout << hex[s[byte] & 15]; // first byte - low nibble

cout << hex[s[!byte] >> 4]; // second byte - high nibble
cout << hex[s[!byte] & 15]; // second byte - low nibble
```

You may extend these preprocessor statements to handle other machines as well. This approach lets you maintain one version of `itoh()` that's portable across different machines.

Declaring a pointer variable

General syntax of pointer declaration is,

*data-type *pointer_name;*

Data type of a pointer must be same as the data type of a variable to which the pointer variable is pointing. **void** type pointer works with all data types, but is not used often used.

Initialization of Pointer variable

Pointer Initialization is the process of assigning address of a variable to **pointer** variable. Pointer variable contains address of variable of same data type. In C language **address operator** `&` is used to determine the address of a variable. The `&` (immediately preceding a variable name) returns the address of the variable associated with it.

```
int a = 10 ;
int *ptr ;    //pointer declaration
ptr = &a ;    //pointer initialization
or,
int *ptr = &a ;    //initialization and declaration together
```

Pointer variable always points to same type of data.

```
float a;  
int *ptr;  
ptr = &a; //ERROR, type mismatch
```

Note: If you do not have an exact address to be assigned to a pointer variable while declaration, It is recommended to assign a NULL value to the pointer variable. A pointer which is assigned a NULL value is called a null pointer.

Dereferencing of Pointer

Once a pointer has been assigned the address of a variable. To access the value of variable, pointer is dereferenced, using the **indirection operator** *.

```
int a,*p;  
a = 10;  
p = &a;  
  
printf("%d",*p); //this will print the value of a.  
  
printf("%d",&a); //this will also print the value of a.  
  
printf("%u",&a); //this will print the address of a.  
  
printf("%u",p); //this will also print the address of a.  
  
printf("%u",&p); //this will print the address of p.
```

Points to remember:

1. While declaring/initializing the pointer variable, * indicates that the variable is a pointer.
2. The address of any variable is given by preceding the variable name with Ampersand '&'.
3. The pointer variable stores the address of a variable. The declaration **int *a** doesn't mean that a is going to contain an integer value. It means that **a** is going to contain the address of a variable storing integer value.
4. To access the value of a certain address stored by a pointer variable, * is used. Here, the * can be read as '**value at**'.

A Simple program to explain pointers:

```
#include  
int main()  
{  
    int i = 10; //normal integer variable storing value 10  
    int *a; //since '*' is used, hence its a pointer variable  
  
    a = &i; //'&' returns the address of the variable i which is stored in the pointer variable a.
```

```
//below, address of variable i, which is stored by a pointer variable a is displayed
printf("Address of variable i is %u\n",a);

//below, '*a' is read as 'value at a' which is 10
printf("Value at an address, which is stored by pointer variable a is %d\n", *a);

return 0;
}
```

Output:

Address of variable i is 2686728 (The address may vary)
Value at an address, which is stored by pointer variable a is 10.

Pointer to a Pointer

Pointers are used to store the address of the variables of specified datatype. But If you want to store the address of a pointer variable, then you again need a pointer to store it. Thus, When one pointer variable stores the address of another pointer variable, it is known as **Pointer to Pointer** variable.

Syntax:

```
int **p1;
```

Here, we have used two indirection operator(*) that stores and points to the address of a pointer variable i.e, int *. If we want to store the address of this variable p1, then the syntax would be:

```
int ***p2
```

Hence, **number of indirection operator(*) - 1** tells you to what type will the current pointer variable will point to.

Simple program to represent Pointer to a Pointer

```
#include <stdio.h>
```

```
int main() {
```

```
    int a=10;
```

```
    int *p1; //this can store the address of variable a
```

```
    int **p2;
```

```
    /*this can store the address of pointer variable p1 only. It cannot store the address of variable a */
```

```
    p1 = &a;
```

```
    p2 = &p1;
```

```
printf("Address of a = %u\n",&a);
printf("Address of p1 = %u\n",&p1);
printf("Address of p2 = %u\n\n",&p2);

printf("Value at the address stored by p2 = %u\n",*p2); //this will give the address of a
printf("Value at the address stored by p1 = %d\n\n",*p1);

printf("Value of **p2 = %d\n", **p2); //read this *(*p2)

/*
    This is not allowed, it will give a compiler time error
    p2 = &a;
    printf("%u",p2);
*/
return 0;
}
```

Output:

Address of a = 2686724

Address of p1 = 2686728

Address of p2 = 2686732

Value at the address stored by p2 = 2686724

Value at the address stored by p1 = 10

Value of **p2 = 10

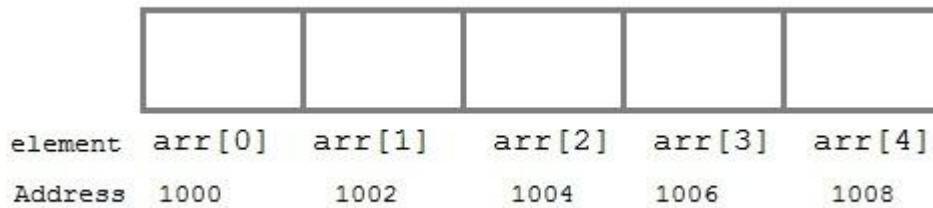
Pointer and Arrays

When an array is declared, compiler allocates sufficient amount of memory to contain all the elements of the array. Base address i.e address of the first element of the array is also allocated by the compiler.

Suppose we declare an array arr,

```
int arr[5]={ 1, 2, 3, 4, 5 };
```

Assuming that the base address of **arr** is 1000 and each integer requires two bytes, the five elements will be stored as follows:



Here variable arr will give the base address, which is a constant pointer pointing to the element, arr[0]. Therefore arr is containing the address of arr[0] i.e 1000. In short, arr has two purpose - it is the name of an array and it acts as a pointer pointing towards the first element in the array.

arr is equal to &arr[0] //by default

We can declare a pointer of type int to point to the array arr.

```
int *p;
p = arr;
or p = &arr[0]; //both the statements are equivalent.
```

Now we can access every element of array arr using p++ to move from one element to another.

NOTE : You cannot decrement a pointer once incremented. p-- won't work.

Pointer to Array

As studied above, we can use a pointer to point to an Array, and then we can use that pointer to access the array. Lets have an example,

```
int i;
int a[5] = { 1, 2, 3, 4, 5 };
int *p = a; // same as int*p = &a[0]
for (i=0; i<5; i++)
{
    printf("%d", *p);
    p++;
}
```

Pointer to Multidimensional Array

A multidimensional array is of form, `a[i][j]`. Lets see how we can make a pointer point to such an array. As we know now, name of the array gives its base address. In `a[i][j]`, `a` will give the base address of this array, even `a + 0 + 0` will also give the base address, that is the address of `a[0][0]` element.

Here is the generalized form for using pointer with multidimensional arrays.

```
*(*(a + i) + j)
```

is same as

```
a[i][j]
```

Pointer and Character strings

Pointer can also be used to create strings. Pointer variables of char type are treated as string.

```
char *str = "Hello";
```

This creates a string and stores its address in the pointer variable `str`. The pointer `str` now points to the first character of the string "Hello". Another important thing to note that string created using char pointer can be assigned a value at **runtime**.

```
char *str;  
str = "hello"; //this is Legal
```

The content of the string can be printed using `printf()` and `puts()`.

```
printf("%s", str);  
puts(str);
```

Notice that `str` is pointer to the string, it is also name of the string. Therefore we do not need to use indirection operator `*`.

Array of Pointers

We can also have array of pointers. Pointers are very helpful in handling character array with rows of varying length.

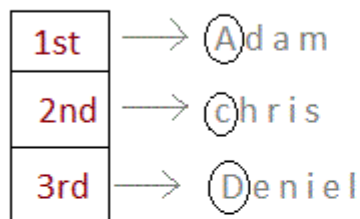
```
char *name[3] = {  
    "Adam",  
    "chris",  
    "Deniel"
```

```

};
//Now see same array without using pointer
char name[3][20] = {
    "Adam",
    "chris",
    "Deniel"
};

```

Using Pointer



char* name[3]

Only 3 locations for pointers, which will point to the first character of their respective strings.

Without Pointer

A	d	a	m			
c	h	r	i	s		
D	e	n	i	e	l	

char name[3][20]

extends till 20 memory locations

In the second approach memory wastage is more, hence it is preferred to use pointer in such cases.

Pointer to Structure

Like we have array of integers, array of pointers etc, we can also have array of structure variables. And to make the use of array of structure variables efficiently, we use **pointers of structure type**. We can also have pointer to a single structure variable, but it is mostly used with array of structure variables.

```

struct Book
{
    char name[10];
    int price;
}

```

```

int main()
{
    struct Book a;    //Single structure variable
}

```



```
struct Book* ptr; //Pointer of Structure type
ptr = &a;

struct Book b[10]; //Array of structure variables
struct Book* p;    //Pointer of Structure type
p = &b;
}
```

Accessing Structure Members with Pointer

To access members of structure with structure variable, we used the dot . operator. But when we have a pointer of structure type, we use arrow -> to access structure members.

```
#include <stdio.h>
int main()
{
    struct my_structure {
        char name[20];
        int number;
        int rank;
    };

    struct my_structure variable = {"StudyTonight",35,1};

    struct my_structure *ptr;
    ptr = &variable;

    printf("NAME: %s\n",ptr->name);
    printf("NUMBER: %d\n",ptr->number);
    printf("RANK: %d",ptr->rank);

    return 0;
}
```

Output:

```
NAME: StudyTonight
NUMBER: 35
RANK: 1
```

Pointer as Function parameter

Pointer as a function parameter list is use to hold address of argument passed during function call. This is also known as **call by reference**. When a function is called by reference any change made to the reference variable will effect the original variable.

Example: Swapping two numbers using Pointer

```
#include <stdio.h>
```

```
void swap(int *a, int *b);
```

```
int main()
{
    int m=10, n=20;
    printf("m = %d\n",m);
    printf("n = %d\n\n",n);

    swap(&m,&n); //passing address of m and n to the swap function
    printf("After Swapping:\n\n");
    printf("m = %d\n",m);
    printf("n = %d",n);
    return 0;
}
```

```
void swap(int *a, int *b)//pointer a and b holds and points to the address of m and n
{
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
}
```

Output:

```
m = 10
n = 20
```

After Swapping:

```
m = 20
n = 10
```

Function returning Pointer

A function can also return a pointer to the calling function. In this case you must be careful, because local variables of function doesn't live outside the function. They have scope only till inside the function. Hence if you return a pointer connected to a local variable, that pointer be will pointing to nothing when function ends.

```
#include <stdio.h>
#include <conio.h>
int* larger(int*, int*);
void main()
{
    int a=15;
    int b=92;
    int *p;
    p=larger(&a, &b);
    printf("%d is larger",*p);
}
```

```
}

int* larger(int *x, int *y)
{
    if(*x > *y)
        return x;
    else
        return y;
}
```

Output:

92 is larger

Safe ways to return a valid Pointer.

1. Either use **argument with functions**. Because argument passed to the functions are declared inside the calling function, hence they will live outside the function called.
2. Or, use **static local variables** inside the function and return it. As static variables have a lifetime until main() exits, they will be available throughout the program.

Pointer to functions

It is possible to declare a pointer pointing to a function which can then be used as an argument in another function. A pointer to a function is declared as follows,

type (***pointer-name**)(*parameter*);

Example :

```
int (*sum)(); //legal declaration of pointer to function
int *sum(); //This is not a declaration of pointer to function
```

A function pointer can point to a specific function when it is assigned the name of the function.

```
int sum(int, int);
int (*s)(int, int);
s = sum;
```

s is a pointer to a function **sum**. Now **sum** can be called using function pointer s with the list of parameter.

```
s (10, 20);
```

Example of Pointer to Function

```
#include <stdio.h>
```

```
#include <conio.h>

int sum(int x, int y)
{
    return x+y;
}

int main( )
{
    int (*fp)(int, int);
    fp = sum;
    int s = fp(10, 15);
    printf("Sum is %d",s);
    getch();
    return 0;
}
```

Output : 25

References as Function Arguments

Function arguments may be references. The formats are

```
Type function_name(Type &);    // prototype

Type function_name(Type & arg)  // definition
{
    function body
}
```

Function calls with arguments initialize a reference and create an alias. Inside the body of the function, the alias arg refers to the corresponding argument in the caller's program. Expressions in function statements with alias names as lvalues (such as assignments or increment operators) modify the argument in the caller. Expressions with aliases as rvalues provide read access to function arguments.

References may appear with default arguments only if you initialize them to statics or globals (see "static" on page 125). Here are several examples.

```
static double dval = 5.6;    // static
char f(double & d = 5.6);    // illegal
char g(double & d = dval);    // legal
```

Why use references instead of pointers in function arguments? To illustrate, let's write a function called bump(), whose job is to increment its integer argument by one. Our first approach doesn't work, but it tells us what we need.

```
inline void bump(int m) {
    ++m;    // increment by one
}
```

```
int num = 10;
bump(num);          // call by value
cout << num << endl; // displays 10
```

We don't increment num because we pass it by value to bump() and modify only a local variable (m). Let's try it again with a pointer argument instead.

```
inline void bump(int *p) {
    ++*p;          // increment by one
}

int num = 10;
bump(&num);        // call by address
cout << num << endl; // displays 11
```

This time we increment num correctly. We pass num by address (&num) so that bump() can modify it using the compact pointer expression ++*p. (This expression increments what p points to.) Although this works, it's easy to confuse the previous expression with *p++, which compiles but does not increment num! (Here, we increment the pointer and not what it points to; see "Compact Pointer Expressions" on page 63.)

These kinds of mistakes can't happen with references because pointer expressions are unnecessary. Here's the code for bump() with references.

```
inline void bump(int &m) {
    ++m;          // increment by one
}

int num = 10;
bump(num);        // call by reference
cout << num << endl; // displays 11
```

This looks like a call by value, except that the function definition makes the compiler pass a reference to bump(), rather than a value. Inside bump(), the compiler generates the code to increment num, using the alias m. Designing bump() with references makes it easy to call (no &) and easy to write (no pointer expressions inside the body of the function).

Constant reference arguments

```
// ref.C - const reference arguments
#include <iostream.h>
```

```
const int MaxBuf = 20;
struct block {
    char buf[MaxBuf];
```

```
    int used;
};
```

```
int main()
```

```

{
void display(const block &);    // prototype for display()
block data;
int i;

data.used = 5;
for (i = 0; i < data.used; i++)    // assign some values
    data.buf[i] = i + 'a'
display(data);                    // call by reference

data.used = MaxBuf;

for (i = 0; i < data.used; i++)
    data.buf[i] = i + 'a';
display(data);                    // call by reference
return 0;
}

void display(const block & blockref) {
for (int i = 0; i < blockref.used; i++)
    cout << blockref.buf[i] << ' ';
cout << endl;
}

$ ref
a b c d e
a b c d e f g h i j k l m n o p q r s t

```

The main() program creates a data structure of type block, and for loops fill its data member array (buf) with characters. Data member used retains a count of the number of valid array elements. The program calls display() twice to print out array elements. Inside display(), we declare blockref a constant reference to block, making calls to display() efficient (no local copies) and safe (no modifications).

KARPAGAM ACADEMY OF HIGHER EDUCATION

(Established Under Section 3 of UGC Act 1956)

COIMBATORE – 641 021

DEPARTMENT OF COMPUTER APPLICATIONS

Batch 2016 – 2019

16MMU304B Programming with C and C++

Part - B

(Each Question carries 2 marks)

1. Define the term derived data type.
2. How to declare simple structures?
3. What is meant by simple structures?
4. What are pointers?
5. Write any two differences between structures and union.
6. List out the main problems with pointers.
7. How to use array as pointers?
8. What is a pointer variable?
9. Write a simple use of pointers.
10. Write a note on dereferencing pointers.

Part - C

(Each Question carries 6 marks)

1. Explain how to manipulate individual members of structures and union.
2. Describe the use of declaring and dereferencing pointers to simple variables.
3. Differentiate between call by value and call by reference with suitable example program.
4. Elaborate in detail the about the concept of pointers and discuss on passing pointers as function arguments with suitable example program.
5. Explain array of structures with example.
6. Discuss the concept of call by function and call by reference in detail.
7. Distinguish between pointer and references with suitable examples.
8. Elaborate how to initialize individual data members as structures with suitable program.
9. Explain how to use references as function arguments and function return values.
10. How to declare and initialize references? Explain with example.
11. Explain the simple use of command line arguments.

KARPAGAM ACADEMY OF HIGHER EDUCATION

(Established Under Section 3 of UGC Act 1956)

COIMBATORE – 641 021

DEPARTMENT OF COMPUTER APPLICATIONS

Batch 2016 – 2019

Subject : Programming with C and C++**Subject Code: 16MMU304B****UNIT - III**

S.No	Questions	opt1	opt2	opt3	opt4	Answer
1	The stream is an _____ between I/O devices and the user.	Trans later	Destination	Intermediator	None	Intermediator
2	If the data is received from the input devices in sequence then it is called_____.	Source stream	Object stream	Destination stream	Input stream	source stream
3	When the data is passed to the output devices it is called_____	Source stream	Object stream	Destination stream	Input stream	destination stream
4	The C++ have a number of stream classes that are used to work with _____ operations.	Console I/O	Console and file	formatted console	file	console and file
5	The data accepted with default setting by I/O function of the language is known as	Formatted	Unformatted	Argumented	extracted	unformatted
6	The ----- system inC++ is designed to work with variety of devices	I/O system	put() system	get() system	Input system	I/O system
7	cin and cout are _____ for input and output of data.	user defined stream	system defined stream	Pre defined stream	undefined stream	system defined stream
8	The data obtained or represented with some manipulators are called _____.	formatted data	unformatted data	extracted data	unextracted data	formatted data
9	The output formats can be controlled with manipulators having the header file as	iostream.h	conio.h	stdlib.h	iomanip.h	iomanip.h
10	The ----- function reads character input into the variable line	getline()	line()	gets()	putline()	getline()
11	The manipulator << endl is equivalent to_____	‘\t’	‘\r’	‘\n’	‘\b’	‘\n’
12	A virtual function must be defined in _____	Friend	enemy	member	class	friend
13	To clear the flags specified ----- function is used	unsetf()	setf()	width()	flag()	unsetf()
14	The function in base class is declared as virtual using the keyword _____	Virtual	Class	Pointer	Structure	virtual
15	Precision() is an _____ format function	Manipulator	Istream	ios	user defined	ios
16	Width of the output field is set using the _____	width()	setf()	unsetf()	line()	width()
17	_____ is used to achieve run time polymorphism	operator overloading	function overloading	inline function	virtual function	virtual function
18	Pointers are used to access _____	Object	Virtual function	Class members	functions	class members
19	The member functions can be refered by using the _____ and _____	dot operator and object	address operator and virtual functions	class and object	functions	dot operator and object
20	The paranthesis are necessary because the dot operator has higher precedence than the _____	dot operator	this	class	indirection operator	indirection operator
21	The ----- flag skip the white space on input ?	ios::skipus	ios::skip	ios::showpoint	ios::skipos	ios::skipus
22	The manipulator Endif is equivalent to -----	"\n"	"\t"	setf()	unsetf()	"\n"
23	When two or more objects are compare inside a member function the result in return is an _____	virtual function	derived class	invoking objects	class	invoking objects
24	Pointers are used as the objects of _____	user defined	derived class	virtual function	object.	derived class
25	Which one of following is an sequence container?	stack	deque	queue	None.	deque
26	The virtual functions are accessed with the help of a pointer declared as ____ to the base class	Class	object	pointer	stream	pointer
27	_____ is achieved when a virtual function is accessed through a pointer to the base class.	run time polymorphism	inheritance	class	constructor	run time polymorphism
28	we cannot have virtual constructors but _____ are allowed	translators	virtual function	virtual destructor	destructor	virtual destructor
29	The virtual functions cannot be _____	class	object	constructors	static members	static members
30	The ----- is the one that is not create objects.	class	abstract	Pointer	member	abstract
31	A ____ is a function declared in a base class that has no definition relative to the base class	Virtual function	pure virtual function	stream	class	pure virtual function
32	A ____ equated to zero is called a pure virtual function.	virtual function	pure virtual function	stream	class.	virtual function
33	Stream and stream classes are used to implement its I/O operations with the _____	the console and disk files	cin and cout	manipulators	streams	the console and disk files
34	A _____ is a sequence of bytes.	Stream	class	object	files	stream
35	The _____ streams automatically open when the program begins its execution	user defined	predefined	input	output	predefined

36	int main() { int x,y=10,z=10; x= (y==z); cout<<x; return 0; } what will be the output ?	0	1	10	error	1
37	Templates enable us to create a range of related----- -----	classes	variables	objects	main() functions	classes
38	The _____ are called as overloaded operators	>> and <<	+ and –	* and &&	– and .	>> and <<
39	The >> operator is overloaded in the _____	istream	ostream	iostream	streams	istream
40	The _____ functions are used to handle the single character I/O operation.	get() and put()	clrscr() and getch()	cin and cout	getch()	get() and put()
41	_____ functions are used to display text more efficiently by using the line oriented i/o functions.	getline() and write()	cin and cout	get() and put()	get()	getline() and write()
42	The getline() reads character input to the _____ line	datatype	function	variable	numeric	variable
43	Which of the following functions give the current size of a string object?	width()	length()	setf()	unsetf()	length()
44	Which of the following are non-mutating algorithms ?	search()	rotate()	count()	both a and c	both a and c
45	By default the floating numbers are printed with _____ after the decimal point.	5 digits	6	7	8	6
46	_____ returns the setting in effect until it is reset	width	precision()	setf()	fill()	precision()
47	Which of the following containers support the random access iterator?	multiset	multimap	vector	both a and b	vector
48	An exception is caused by -----	run time error	syntax error	hardware problem	both b and c	run time error
49	What is the error in the program? Class test { virtual void display(); }	no error	test class contain data member	display() should be defined	both b and c	no error
50	A ---- and ----- use similar syntax	structure and class	class and object	both a&b	none of the above	structure and class
51	The flag formatted for the octal base is _____	ios::dec	ios::hex	ios::fixwd	ios::oct	ios::oct
52	The flag is formatted with _____ arguments.	1	2	3	4	1
53	The bit field is formatted with _____ arguments.	1	2	3	4	2
54	_____ flush all streams after insertion	ios::stdio	ios::shoebase	ios::showpoint	ios:: unitbuf	ios::unitbuf
55	_____ is used as base indicator on output.	ios::stdio	ios::showbase	ios::showpoint	d ios:: unitbuf	ios::showbase
56	In -----,data can be quickly inserted or deleted at either end	deque	queue	vector	endif	deque
57	_____ returns the previous format state.	ios member function	manipulator	class	a and b	ios member function
58	The bitfield used for fixed point notation is _____	ios::floatfield	ios::adjustfield	ios::basefield	none	ios::floatfield
59	The statement -----;rethrows an exception	throw	rethrow	default	exception	throw

KARPAGAM ACADEMY OF HIGHER EDUCATION

(Established Under Section 3 of UGC Act 1956)

COIMBATORE – 641 021

DEPARTMENT OF COMPUTER APPLICATIONS

Batch 2016-2019

Subject : PROGRAMMING WITH C and C++

Sub.code : 16MMU304B

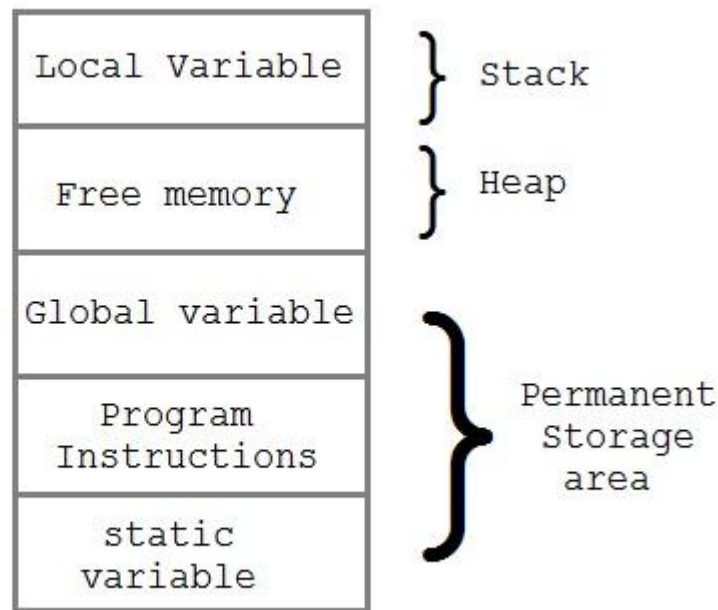
Dynamic Memory Allocation

The process of allocating memory at runtime is known as **dynamic memory allocation**. Library routines known as "memory management functions" are used for allocating and freeing memory during execution of a program. These functions are defined in **stdlib.h**.

Function	Description
malloc()	allocates requested size of bytes and returns a void pointer pointing to the first byte of the allocated space
calloc()	allocates space for an array of elements, initialize them to zero and then returns a void pointer to the memory
free	releases previously allocated memory
realloc	modify the size of previously allocated space

Memory Allocation Process

Global variables, **static** variables and program instructions get their memory in **permanent** storage area whereas **local** variables are stored in a memory area called **Stack**. The memory space between these two region is known as **Heap** area. This region is used for dynamic memory allocation during execution of the program. The size of heap keep changing.



Allocating block of Memory

malloc() function is used for allocating block of memory at runtime. This function reserves a block of memory of given size and returns a pointer of type void. This means that we can assign it to any type of pointer using typecasting. If it fails to allocate enough space as specified, it returns a NULL pointer.

Syntax:

```
void* malloc(byte-size)
```

Example using malloc() :

```
int *x;  
x = (int*)malloc(50 * sizeof(int)); //memory space allocated to variable x  
free(x); //releases the memory allocated to variable x
```

calloc() is another memory allocation function that is used for allocating memory at runtime. **calloc** function is normally used for allocating memory to derived data types such as **arrays** and **structures**. If it fails to allocate enough space as specified, it returns a NULL pointer.

Syntax:

```
void *calloc(number of items, element-size)
```

Example using calloc() :

```
struct employee  
{
```

```
char *name;
int salary;
};
typedef struct employee emp;
emp *e1;
e1 = (emp*)calloc(30,sizeof(emp));
```

realloc() changes memory size that is already allocated dynamically to a variable.

Syntax:

```
void* realloc(pointer, new-size)
```

Example using realloc() :

```
int *x;
x=(int*)malloc(50 * sizeof(int));
x=(int*)realloc(x,100); //allocated a new memory to variable x
```

Difference between malloc() and calloc()

calloc()

calloc() initializes the allocated memory with 0 value.

Number of arguments is 2

malloc()

malloc() initializes the allocated memory with garbage values.

Number of argument is 1

Syntax :

```
(cast_type *)calloc(blocks
size_of_block);
```

Syntax :

```
' (cast_type *)malloc(Size_in_bytes);
```

Program to represent Dynamic Memory Allocation(using calloc())

```
#include <stdio.h>
```

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

```
int main()
```

```
{
```

```
    int i, n;
```

```
    int *element;
```

```
    printf("Enter total number of elements: ");
```

```
    scanf("%d", &n);
```

```
    element = (int*) calloc(n,sizeof(int)); /*returns a void pointer(which is type-casted to int*)
pointing to the first block of the allocated space*/
```

```
    if(element == NULL)//If it fails to allocate enough space as specified, it returns a NULL
pointer.
```

```
    {
```

```
        printf("Error.Not enough space available");
```

```
        exit(0);
    }

    for(i=0;i<n;i++)
        scanf("%d",element+i); //storing elements from the user in the allocated space

    for(i=1;i<n;i++)
    {
        if(*element > *(element+i))
            *element = *(element+i);
    }

    printf("Smallest element is %d",*element);

    return 0;
}
```

Output:

```
Enter total number of elements: 5
4 2 1 5 3
Smallest element is 1
```

Memory Allocation:

There are three types of allocation — static, automatic, and dynamic.

Static Allocation means, that the memory for your variables is allocated when the program starts. The size is fixed when the program is created. It applies to global variables, file scope variables, and variables qualified with static defined inside functions.

Automatic memory allocation occurs for (non-static) variables defined inside functions, and is usually stored on the *stack* (though the C standard doesn't mandate that a stack is used). You do not have to reserve extra memory using them, but on the other hand, have also limited control over the lifetime of this memory. E.g: automatic variables in a function are only there until the function finishes.

```
void func() {
    int i; /* `i` only exists during `func` */
}
```

Dynamic memory allocation is a bit different. You now control the exact size and the lifetime of these memory locations. If you don't free it, you'll run into memory leaks, which may cause your application to crash, since at some point of time, system cannot allocate more memory.

```
int* func() {
    int* mem = malloc(1024);
    return mem;
}
```

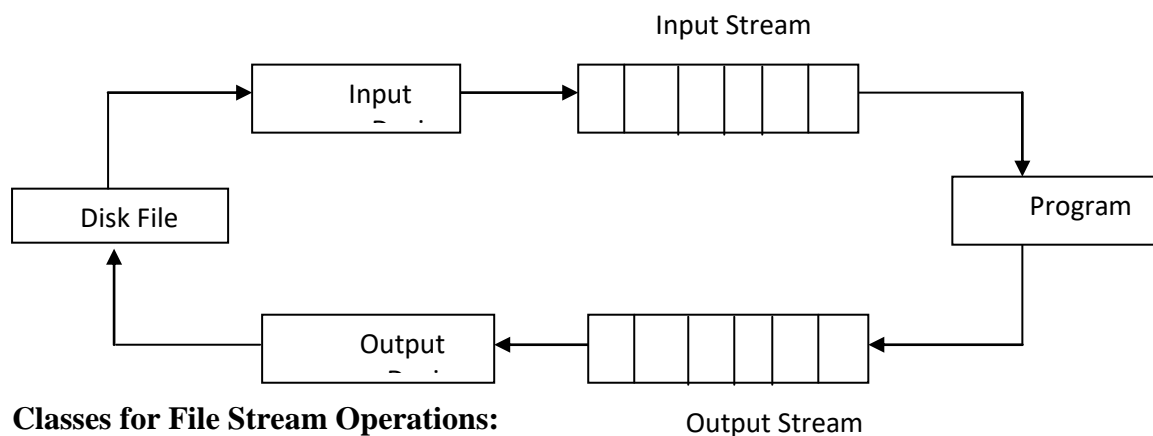
```
}
```

```
int* mem = func(); /* still accessible */
```

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:

- 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 istream 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
ios::nocreate	Open fails if the file does not exist
ios::noreplace	Opens fails if the file already exist
ios::out	Open file for writing only
ios::trunc	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)`

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 task 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 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

KARPAGAM ACADEMY OF HIGHER EDUCATION

(Established Under Section 3 of UGC Act 1956)

COIMBATORE – 641 021

DEPARTMENT OF COMPUTER APPLICATIONS

Batch 2016 – 2019

16MMU304B Programming with C and C++

Part - B

(Each Question carries 2 marks)

1. What are streams?
2. Discuss the use of delete operator.
3. How to malloc and calloc?
4. What is meant by free functions?
5. Write the use of new operator.
6. What is random access in files?
7. Define preprocessor directives.
8. Write about static memory allocation.
9. Write on dynamic memory allocation.
10. What is a macro?
11. List the use of ifstream and ofstream classes.
12. List the use of fstream header files.
13. Write the use of delete operator.

Part - C

(Each Question carries 6 marks)

1. What is meant by memory allocation? Explain the dynamic memory allocation in detail.
2. Write a C++ program to Read and Write Text files with fstream header.
3. Elaborate the use of fstream header file and ifstream file in c++ with a suitable program.
4. Discuss about storage of variables in dynamic memory allocation.
5. Explain about the use of New and Delete operators in memory.
6. Differentiate between static and dynamic memory allocation in detail.
7. Explain in detail the preprocessor directives.
8. Explain in detail opening and closing of a file with an appropriate example.
9. Elaborate in detail the storage of variables in static and dynamic memory allocation.
10. Describe the concept of reading and writing text files in c++.
11. Differentiate between static and dynamic memory allocation.

KARPAGAM ACADEMY OF HIGHER EDUCATION

(Established Under Section 3 of UGC Act 1956)

COIMBATORE – 641 021

DEPARTMENT OF COMPUTER APPLICATIONS

Batch 2016 – 2019

Subject : Programming with C and C++**Subject Code: 16MMU304B****UNIT - IV**

S.No	Questions	opt1	opt2	opt3	opt4	Answer
1	A _____ is a collection of related data stored in a particular	Field	File	Row	Vector	file
2	File streams act as an _____ between programs and files.	interface	converter	translator	operator	interface
3	Ifstream, Ofstream, Fstream are derived form _____.	iostream	ostream	streambuff	fstreambase	fstreambase
4	Classes designed to manage the _____ files are declared	random	sequential	disk	tape	disk
5	A file can be opened in ----- ways	2	3	4	5	2
6	_____ inherits get(), getline(), read(), seekg(), and tellg()	conio	ifstream	fstream	iostream	ifstream
7	The ----- condition is necessary to preventing any further attempt to read data from the file	detection end-of-file	open() file	end-of-file	none of the above	detection end-of-file
8	_____ inherits all functions from istream and ostream	conio	ofstream	fstream	ifstream	fstream
9	The file mode parameter for opening a binary file is	ios::ate	ios::hex	ios::dec	ios::binary	ios::binary
10	_____ is the file mode parameter for go to end of file on	ios::ate	ios::app	ios::del	ios::end	ios::ate
11	The file mode parameter for writing only onto the file is	ios::in	ios::app	ios::ate	ios::out	ios::out
12	Opening a file in ios::out mode also opens it in the _____	ios::trunc	ios::create	ios::create	ios::ate	ios::trunc
13	Both _____ and _____ take us to the end of the file	ios::ate, ios::create	ios::trunk, ios::ate	ios::app, ios::ate	ios::app, ios::out	ios::app,ios::ate
14	The parameter _____ can be used only with the files capable of output.	ios::ate	ios::app	ios::in	ios::create	ios::app
15	The parameter ios::app can be used only with the files capable	input	input and output	append	output	output
16	The eof () stands for _____.	end of file	error opening file	error of file	none of the above	end of file
17	Command line arguments are used with _____ function	main()	member function	with all function	none of the above	main()
18	The close() function _____.	closes the file	closes all files opened	closes only read mode file	none	closes the file
19	The write() function writes _____.	single character	Binary data	string	none of these	Binary data
20	The _____ function shifts the associated files input	seekg()	seekp()	tellg()	tellp().	seekg()
21	The _____ function shifts the associated files output	seekg()	seekp()	tellg()	tellp().	tellg()
22	The object of fstream class provides _____operation	both read and write	read only	write only	none of the above.	both read and write
23	Each file has ----- associated pointers which is known as file	1	3	5	2	2
24	When a file is opened read or write mode a file pointer is set	beginning	end	middle	none	beginning
25	The ----- seek call indicates that it stay at the current position	fout.seekg(o,ios::cur)	fout.seekg(m,ios::cur)	fout.seekg(-m,ios::cur)	all the above	fout.seekg(o,ios::cur)
26	The constructor of this class requires _____file name and	ofstream	ifstream	fstream	all the above	fstream
27	Templates are suitable for _____ data type.	any	basic	derived	all the above	basic
28	Templates can be declared using the keyword _____	class	template	try	none	template
29	Templates is also called as _____ class.	generic	container	virtual	base	generic
30	The ----- is a routine task in the maintenance of any data	updating	dowmloading	random	none.	updating
31	Select the correct Template definition _____ .	template <class T>	class <template T>	template <T>	template class <T>.	template<class T>
32	Function Templates are normally defined _____ .	in main function	globally	in a class	anywhere	in a class
33	The ----- arguments are typed by the user and delimited by a	command-line	default	template	counter arguments	command-line
34	_____ section defines all symbolic constants	link	documentation	definition	subprogram	definition
35	_____ line should not end with semicolon	# define	variable declaration	assignment statements	function calling	# define
36	The file function fclose returns the value _____ if he file is	0	1	TRUE	FALSE	1
37	First parameter in the command line is always _____	file name	argument count	argument vector	size	argument count
38	A _____ is placed on the disk where a group of related data is	struture	union	bit field	file	file
39	In _____ mode the existing file is opened for reading only	r	w	a	f	r
40	In _____ mode the file can be opened for writing only	r	w	a	f	w
41	In _____ mode the file can be opened for appending data to it	r	w	a	f	a
42	The getc function will return an _____,when end of the file	BOF	EOF	SOF	FOF	EOF
43	In fseek function value of offset should be _____ to move the	0	1	2	3	0
44	In fseek function value of offset should be _____ to move the	0	1	2	3	1
45	In fseek function value of offset should be _____ to move the	0	1	2	3	2
46	_____ is a parameter supplied to a program when the program is invoked	argument	parameter	command line argument	values	command line argument
47	Command line argument is supplied to the program when it is	invoked	developed	compiled	stored	invoked
48	The file mode _____ is used to read and append some data into an existing file from end of the file	"r"	"r-"	"r+"	all	"r+"
49	A _____ file is a collection of ASCII characters, with end of line markers and end of file markers	program	binary	image	text	text

50	The files opened with mode “w” allows	reading and writing	reading only	writing only	reading and appending	writing only
51	The files opened with mode “a+” allows	reading and writing	reading only	writing only	reading and appending	reading and appending
52	The _____ function can be used to test for an end of file	eof()	feof()	eol()	EOF	feof()
53	The negative integer constant EOF indicates the ____	end of line	new line	end of file	minimum integer range	end of file
54	fclose() function is used to close	editor	program	all the above	file	file
55	File mode must be specified while _____	opening a file	reading a file	writing a file	closing a file	opening a file
56	_____ function is used to write a string into an ASCII file	fwrite()	writes()	puts()	fputs()	fputs()
57	_____ is used to read a number of items from the file	printf()	scanf()	fprintf()	fscanf()	fscanf()
58	_____ function is used to report the status of the file and returns a non zero integer if an error has been detected	fstatus()	ferror()	fnull()	ifzero()	ferror()
59	fseek, ftell and rewind functions are used with _____ files	sequential files	indexed files	random access files	all files	random access files
60	_____ returns the current position of the file pointer in a file	pos()	fseek()	ftell()	fposition()	ftell()
61	_____ is used to move the file pointer to the desired location	pos()	fseek()	ftell()	fposition()	fseek()
62	_____ takes the filepointer and reset the position to the	pos()	fseek()	ftell()	rewind()	rewind()
63	Which of the following functions will take only the file	fclose()	ferror()	rewind()	all of the above	all of the above
64	_____ function is used to read some bytes from the binary	read()	readln()	readf()	fread()	fread()
65	_____ function is used to write some bytes into a binary	writebytes()	fwrite()	writef()	putfile()	fwrite()
66	main() function takes _____ number of arguments	one	two	three	any	two
67	The _____ in the main(argc, argv) function represents an array of character pointers that points to the command line	argc	argv	args	cptr	argv
68	In main(argc,argv),the variable argc ____	counts the number of arguments in command line	b) counts the number of functions in a program	arranges the argument in the command line	sets a pointer to the argument in the command line	counts the number of arguments in command line
69	In the command line argv[0] points to the _____	program under execution	first argument after the program	the beginning of the program file	all the elements in the arguments vector	program under execution
70	Command line arguments are used to accept argument from	command prompt of operating system	through scanf() statement	through printf() statement	through gets function	command prompt of operating system
71	The malloc() function	allocates memory and not returns a pointer	allocates memory and returns a pointer to the first byte of it	changes the size of allocated memory	deallocates or frees the memory	allocates memory and returns a pointer to the first byte of it
72	The _____ processes the source code before it passes	interpreter	preprocessor	linker	assembler	preprocessor
73	Preprocessor directives must be present	before the main () function	after the main() function	at the end of the program	anywhere in the program body	before the main () function
74	When files are included using #include<filename> ,then the file is searched in_____	standard library only	current directory only	in all directories	current directories & in standard library	standard library only
75	FILE *fp; fp=fopen(“ fn”,”m”); In the above syntax fp is a _____	file name	file variable	pointer to data type FILE	pointer to function fopen()	pointer to data type FILE
76	_____ is analogous to getchar() function and reads a	getch()	gets()	getc()	getw()	getc()
77	_____ is functions used to write integers into a file	putw()	puts	puti()	putc	putw()

KARPAGAM ACADEMY OF HIGHER EDUCATION

(Established Under Section 3 of UGC Act 1956)

COIMBATORE – 641 021

DEPARTMENT OF COMPUTER APPLICATIONS

Batch 2016-2019

Subject : PROGRAMMING WITH C and C++

Sub.code : 16MMU304B

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

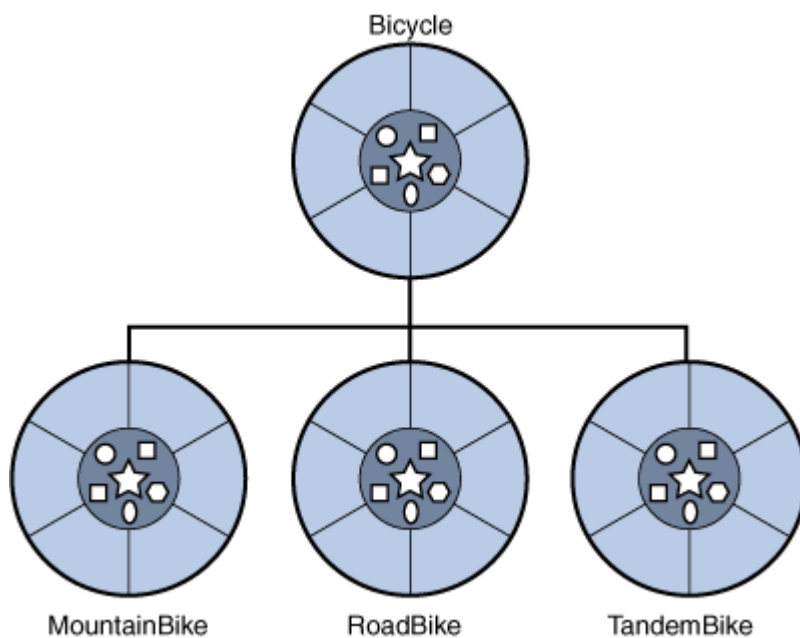
Data abstraction and Encapsulation

- 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 (a suppression of detail).
- 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 than 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.

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:**

```
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:

```
RT class_name::function_name(arg list)
{
    function body;
}
```

Example:

```
void book::getpage()
{
    cout<<"Enter the page No:";
    cin>>pgno;
}
```

Constructors

A class **constructor** is a special member function of a class that is executed whenever we create new objects of that class.

A constructor will have exact same name as the class and it does not have any return type at all, not even void. Constructors can be very useful for setting initial values for certain member variables.

Unlike normal functions, constructors have specific rules for how they must be named:

- 1) Constructors should always have the same name as the class (with the same capitalization)
- 2) Constructors have no return type (not even void)

A constructor that takes no parameters (or has all optional parameters) is called a **default constructor**.

Default Constructor-: A constructor that accepts no parameters is known as default constructor. If no constructor is defined then the compiler supplies a default constructor.

```
student :: student()
```

```
{
    rollno=0;
    marks=0.0;
}
```

Parameterized Constructor -: A constructor that receives arguments/parameters, is called parameterized constructor.

```
student :: student(int r)
```

```
{
```

```
rollno=r;
}
```

Example

```
#include<iostream>
#include<conio.h>
using namespace std;
class Example
{
    // Variable Declaration
    int a,b;
public:

    //Constructor
    Example()
    {
        // Assign Values In Constructor
        a=10;
        b=20;
        cout<<"Im Constructor\n";
    }
    void Display()
    {
        cout<<"Values : "<<a<<"\t"<<b;
    }
};

int main()
{
    Example Object;
    // Constructor invoked.
    Object.Display();

    // Wait For Output Screen
```

```
    getch();  
    return 0;  
}
```

Sample Output

In Constructor

Values :10 20

Example

```
#include <iostream>  
using namespace std;  
class Line  
{ public:  
    void setLength( double len );  
    double getLength( void );  
    Line(); // This is the constructor  
private:  
    double length;  
};  
  
// Member functions definitions including constructor  
Line::Line(void)  
{  
    cout << "Object is being created" << endl;  
}  
void Line::setLength( double len )  
{  
    length = len;  
}  
double Line::getLength( void )  
{  
    return length;  
}  
  
// Main function for the program
```

```
int main( )
{
    Line line;
    // set line length
    line.setLength(6.0);
    cout << "Length of line : " << line.getLength() << endl;
    return 0;
}
```

When the above code is compiled and executed, it produces following result:

```
Object is being created
Length of line : 6
```

Special Characteristics of Constructors

1. They should be declared in the public section.
2. They are invoked automatically when the objects are created
3. They do not have return types, not even void and therfor and they cannot return values.
4. They cannot be inherited, though a derived class can call the base class constructor
5. like other c++ functions, they can have default arguments.
6. constructors cannot be virtual
7. we cannot refer to their addresses.
8. An object with a constructor cannot be used as a member of a union.
9. They make implicit calls to the operators new and delete when memory allocation is required.

Multiple Constructors in a class

Like functions, it is also possible to overload constructors. A class can contain more than one constructor. This is known as constructor overloading. All constructors are define with the same name as the class. All the constructors contain different number of arguments. Depending upon number of arguments, the compiler executes appropriate constructor.

Constructor is automatically called when object(instance of class) create. It is special member function of the class. Which constructor has arguments that's called Parameterized Constructor.

- One Constructor overload another constructor is called Constructor Overloading
- It has same name of class.

- It must be a public member.
- No Return Values.
- Default constructors are called when constructors are not defined for the classes.

Syntax

```
class class-name
```

```
{ Access Specifier:
```

```
    Member-Variables
```

```
    Member-Functions
```

```
public:
```

```
    class-name()
```

```
    {
```

```
        // Constructor code
```

```
    }
```

```
    class-name(variables)
```

```
    {
```

```
        // Constructor code
```

```
    }
```

```
    ... other Variables & Functions
```

```
}
```

Example Program

```
#include<iostream>
```

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

```
using namespace std;
```

```
class Example    {
```

```
    // Variable Declaration
```

```
    int a,b;
```

```
    public:
```

```
    //Constructor wuithout Argument
```

```
    Example()    {
```

```
        // Assign Values In Constructor
```

```
        a=50;
```

```
        b=100;
```

```
        cout<<"\nIm Constructor";
```



```
}  
//Constructor with Argument  
Example(int x,int y)    {  
    // Assign Values In Constructor  
    a=x;  
    b=y;  
    cout<<"\nIm Constructor";  
}  
void Display()  {  
    cout<<"\nValues : "<<a<<"\t"<<b;  
}  
};  
int main()    {  
    Example Object(10,20);  
    Example Object2;  
    // Constructor invoked.  
    Object.Display();  
    Object2.Display();  
    // Wait For Output Screen  
    getch();  
    return 0;  
}
```

Sample Output

Im Constructor

Im Constructor

Values :10 20

Values :50 100

Constructors Overloading are used to increase the flexibility of a class by having more number of constructor for a single class. By have more than one way of initializing objects can be done using overloading constructors.

Example:

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

```

{   public:
    int x;
    int y;
    Overclass() { x = y = 0; }
    Overclass(int a) { x = y = a; }
    Overclass(int a, int b) { x = a; y = b; }
};

int main()
{   Overclass A;
    Overclass A1(4);
    Overclass A2(8, 12);
    cout << "Overclass A's x,y value:: " <<
        A.x << " , " << A.y << "\n";
    cout << "Overclass A1's x,y value:: " <<
        A1.x << " , " << A1.y << "\n";
    cout << "Overclass A2's x,y value:: " <<
        A2.x << " , " << A2.y << "\n";
    return 0;
}

```

Result:

Overclass A's x,y value:: 0 , 0

Overclass A1's x,y value:: 4 ,4

Overclass A2's x,y value:: 8 , 12

In the above example the constructor "Overclass" is overloaded thrice with different initialized values

Constructors with default arguments

Like functions, it is also possible to declare constructors with default arguments. Consider the following example.

```
power (int 9, int 3);
```

In the above example, the default value for the first argument is nine and three for second.

```
power p1 (3);
```

In this statement, object p1 is created and nine raise to 3 expression n is calculated. Here, one argument is absent hence default value 9 is taken, and its third power is calculated. Consider the example on the above discussion given below.

Write a program to declare default arguments in constructor. Obtain the power of the number.

```
#include <iostream.h>
#include <conio.h>
#include <math.h>
class power
{
    private:
        int num;
        int power;
        int ans;
    public :
power (int n=9,int p=3); //
declaration of constructor with default arguments
    void show( )
    {
        cout <<"\n"<<num <<" raise to "<<power <<" is " <<ans;
    }
};
power :: power (int n,int p )
{
    num=n;
    power=p;
    ans=pow(n,p);
}
main( )
{
    clrscr( );
    class power p1,p2(5);
    p1.show( );
```

```
p2.show( );  
return 0;  
}
```

Copy Constructor

Copy Constructor- A constructor that initializes an object using values of another object passed to it as parameter, is called copy constructor. It creates the copy of the passed object.

```
student :: student(student &t)
```

```
{  
    rollno = t.rollno;  
}
```

```
#include<iostream>
```

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

```
class Example
```

```
{  
    // Variable Declaration  
    int a,b;  
    public:  
    //Constructor with Argument  
    Example(int x,int y)  
    {  
        // Assign Values In Constructor  
        a=x;  
        b=y;  
        cout<<"\nIm Constructor";  
    }  
    void Display()  
    {  
        cout<<"\nValues : "<<a<<"\t"<<b;  
    }  
};  
int main()
```

```
{    Example Object(10,20);
    //Copy Constructor
    Example Object2=Object;
    // Constructor invoked.
    Object.Display();
    Object2.Display();
    // Wait For Output Screen
    getch();
    return 0;
}
```

Sample Output

In Constructor

Values :10 20

Values :10 20

Simple Program for Copy Constructor Using C++ Programming

```
#include<iostream.h>
#include<conio.h>
class copy
{
    int var,fact;
public:
    copy(int temp)
    {   var = temp;
    }
    double calculate()
    {   fact=1;
        for(int i=1;i<=var;i++)
        {
            fact = fact * i;
        }
        return fact;
    }
};
```

```
void main()
{ clrscr();
  int n;
  cout<<"\n\tEnter the Number : ";
  cin>>n;
  copy obj(n);
  copy cpy=obj;
  cout<<"\n\t"<<n<<" Factorial is:"<<obj.calculate();
  cout<<"\n\t"<<n<<" Factorial is:"<<cpy.calculate();
  getch();
}
```

Output:

Enter the Number: 5

Factorial is: 120

Factorial is: 120

DESTRUCTORS

The *destructor* fulfills the opposite functionality. **It is automatically called when an object is destroyed**, either because its scope of existence has finished (for example, if it was defined as a local object within a function and the function ends) or because it is an object dynamically assigned and it is released using the operator delete.

The destructor must have the same name as the class, but preceded with a tilde sign (~) and it must also return no value.

The use of destructors is especially suitable when an object assigns dynamic memory during its lifetime and at the moment of being destroyed we want to release the memory that the object was allocated.

// example on constructors and destructors

```
#include <iostream>
```

```
using namespace std;
```

```
class CRectangle {
```

```
    int *width, *height;
```

```
public:
```

```
    CRectangle (int,int);
```

```
~CRectangle ();  
int area () {return (*width * *height);} }  
};  
CRectangle::CRectangle (int a, int b) {  
    width = new int;  
    height = new int;  
    *width = a;  
    *height = b;  
}  
CRectangle::~~CRectangle () {  
    delete width;  
    delete height;  
}  
int main () {  
    CRectangle rect (3,4), rectb (5,6);  
    cout << "rect area: " << rect.area() << endl;  
    cout << "rectb area: " << rectb.area() << endl;  
    return 0;  
}
```

Output :

rect area: 12

rectb area: 30

Following example explain the concept of destructor:

```
#include <iostream>  
using namespace std;  
class Line  
{ public:  
    void setLength( double len );  
    double getLength( void );  
    Line(); // This is the constructor declaration  
    ~Line(); // This is the destructor: declaration  
private:  
    double length;
```

```
};  
// Member functions definitions including constructor  
Line::Line(void)  
{   cout << "Object is being created" << endl;  
}  
Line::~~Line(void)  
{   cout << "Object is being deleted" << endl;  
}  
void Line::setLength( double len )  
{   length = len;  
}  
double Line::getLength( void )  
{   return length;  
}  
// Main function for the program  
int main( )  
{   Line line;  
    // set line length  
    line.setLength(6.0);  
    cout << "Length of line : " << line.getLength() << endl;  
    return 0;  
}
```

When the above code is compiled and executed, it produces following result:

Object is being created

Length of line : 6

Object is being deleted

A destructor is a member function having same name as that of its class preceded by ~(tilde) sign and which is used to destroy the objects that have been created by a constructor. It gets invoked when an object's scope is over.

```
~student() { }
```

Example : In the following program constructors, destructor and other member functions are defined inside class definitions. Since we are using multiple constructor in class so this example also illustrates the concept of constructor overloading


```
#include<iostream.h>
class student //specify a class
{
private :
    int rollno; //class data members
    float marks;
public:
    student() //default constructor
    {
        rollno=0;
        marks=0.0;
    }
    student(int r, int m) //parameterized constructor
    {
        rollno=r;
        marks=m;
    }
    student(student &t) //copy constructor
    {
        rollno=t.rollno;
        marks=t.marks;
    }
    void getdata() //member function to get data from user
    {
        cout<<"Enter Roll Number : ";
        cin>>rollno;
        cout<<"Enter Marks : ";
        cin>>marks;
    }
    void showdata() // member function to show data
    {
        cout<<"\nRoll number: "<<rollno<<"\nMarks: "<<marks;
    }
}
```

```
~student() //destructor
{
}

};

int main()
{
    student st1; //default constructor invoked
    student st2(5,78); //parameterized constructor invoked
    student st3(st2); //copy constructor invoked
    st1.showdata(); //display data members of object st1
    st2.showdata(); //display data members of object st2
    st3.showdata(); //display data members of object st3
    return 0;
}
```

Templates

- Templates are 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 substituted 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

- 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 except 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();  
}
```

Example

```
#include <iostream>
using namespace std;
template <class T>
class mypair {
    T a, b;
public:
    mypair (T first, T second)
        {a=first; b=second;}
    T getmax ();
};
template <class T>
T mypair<T>::getmax ()
{
    T retval;
    retval = a>b? a : b;
    return retval;
}
int main () {
    mypair <int> myobject (100, 75);
    cout << myobject.getmax();
    return 0;
}
o/p
100
```

Function templates

Templates are the foundation of generic programming, which involves writing code in a way that is independent of any particular type.

A template is a blueprint or formula for creating a generic class or a function. The library containers like iterators and algorithms are examples of generic programming and have been developed using template concept.

There is a single definition of each container, such as **vector**, but we can define many different kinds of vectors for example, **vector <int>** or **vector <string>**.

You can use templates to define functions as well as classes

- 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<<"\n m and n before swap: "<<m<<" "<<n;
  swap(m,n);
```

```

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

```

The following is the example of a function template that returns the maximum of two values:

```

#include <iostream>
#include <string>
using namespace std;
template <typename T>
inline T const& Max (T const& a, T const& b)
{  return a < b ? b:a;
}
int main ()
{  int i = 39;
   int j = 20;
   cout << "Max(i, j): " << Max(i, j) << endl;
   double f1 = 13.5;
   double f2 = 20.7;
   cout << "Max(f1, f2): " << Max(f1, f2) << endl;
   string s1 = "Hello";
   string s2 = "World";
   cout << "Max(s1, s2): " << Max(s1, s2) << endl;
   return 0;}

```

If we compile and run above code, this would produce the following result:

Max(i, j): 39

Max(f1, f2): 20.7

Max(s1, s2): World

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
}
```

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;
}
```

Function overloading

Function overloading in C++: C++ program for function overloading. Function overloading means two or more functions can have the same name but either the number of arguments or the data type of arguments has to be different. Return type has no role because function will return a value when it is called and at compile time compiler will not be able to determine which function to call. In the first example in our code we make two functions one for adding two integers and other for adding two floats but they have same name and in the second program we make two functions with identical names but pass them different number of arguments. Function overloading is also known as compile time polymorphism.

```
#include <iostream>
using namespace std;
/* Function arguments are of different data type */
long add(long, long);
float add(float, float);
int main()
{
    long a, b, x;
    float c, d, y;
    cout << "Enter two integers\n";
    cin >> a >> b;
    x = add(a, b);
    cout << "Sum of integers: " << x << endl;
    cout << "Enter two floating point numbers\n";
    cin >> c >> d;
    y = add(c, d);
    cout << "Sum of floats: " << y << endl;
    return 0;
}
long add(long x, long y)
{
    long sum;
```

```
    sum = x + y;
    return sum;
}
float add(float x, float y)
{
    float sum;
    sum = x + y;
    return sum;
}
```

In the above program, we have created two functions "add" for two different data types you can create more than two functions with same name according to requirement but making sure that compiler will be able to determine which one to call. For example you can create add function for integers, doubles and other data types in above program. In these functions you can see the code of functions is same except data type, C++ provides a solution to this problem we can create a single function for different data types which reduces code size which is via templates.

```
#include <iostream>
using namespace std;
/* Number of arguments are different */
void display(char []); // print the string passed as argument
void display(char [], char []);
int main()
{
    char first[] = "C programming";
    char second[] = "C++ programming";
    display(first);
    display(first, second);
    return 0;
}
void display(char s[])
{
    cout << s << endl;
}
```

```
void display(char s[], char t[])
{
    cout << s << endl << t << endl;
}
```

Output of program:

C programming

C programming

C++ programming

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; }
```

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.

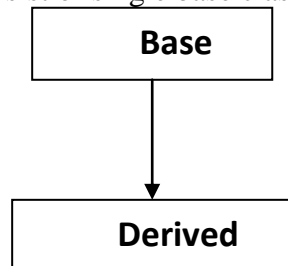
- The visibility-mode is optional, if presents private or public or protected access specifies can be specified
- By default visibility-mode is private.
- The visibility-mode specifies whether the features of the base class are privately derived or publicly derived.

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:

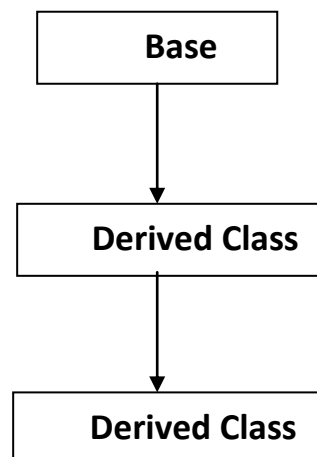
```
#include<iostream.h>
class Rectangle
{
    private:
        float length ; // This can't be inherited
    public:
        float breadth ; // The data and member functions are inheritable
        void Enter_lb(void)
        {
            cout << "\n Enter the length of the rectangle : ";
            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 class visibility	Derived class visibility		
	public derivation	private derivation	protected derivation
private	Not Inherited	Not Inherited	Not Inherited
protected	protected	private	protected
public	public	private	protected

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( )
```

```

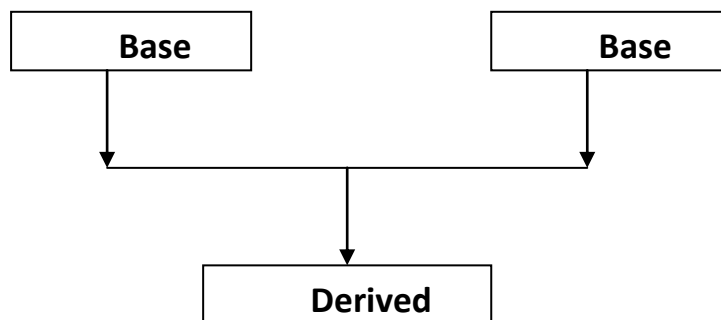
    {
        Derive1::DisplayData( );

        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);
    }
};

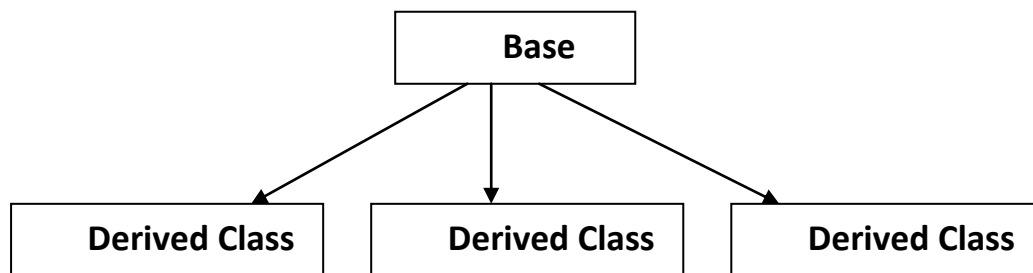
class Cylinder : public Circle, public Rectangle // Derived class, inherited
{ // from classes Circle & Rectangle
public:
    void volume_cy(void)
    {
        cout << "\t The volume of the cylinder is: " << (22/7* radius*radius*length) ;
    }
};

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
}

```

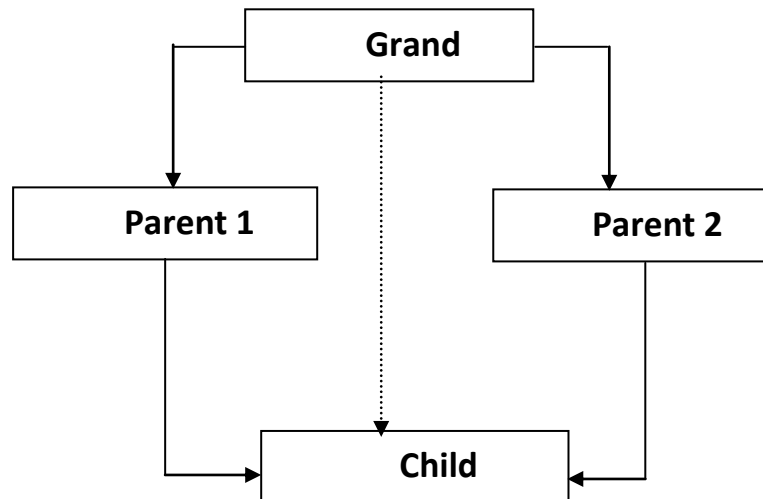
Example:

```

#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.

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;  
}
```

KARPAGAM ACADEMY OF HIGHER EDUCATION

(Established Under Section 3 of UGC Act 1956)

COIMBATORE – 641 021

DEPARTMENT OF COMPUTER APPLICATIONS

Batch 2016 – 2019

16MMU304B Programming with C and C++

Part - B

(Each Question carries 2 marks)

1. What is meant by inheritance?
2. What is meant by multilevel inheritance?
3. Define the terms constructor and destructor.
4. List out the principles of object oriented programming.
5. Write a note on exceptional handling.
6. List out the basics of exceptional handling.
7. What are virtual functions?
8. How to look at an operator as a function call.
9. Define the term pure virtual functions.
10. What are overloading operators?
11. How to restrict exceptions?
12. How to rethrow exceptions?
13. What is meant by polymorphism?
14. Write a note on template classes.

Part - C

(Each Question carries 6 marks)

1. Define the terms virtual functions and pure virtual functions with a suitable program.
2. Describe in detail the need of overloading functions and operators with a suitable program.
3. Write a note on template classes and their uses.
4. Elaborate in detail the concept of multilevel inheritance with a suitable program.
5. Elaborate the concept of operator overloading with a suitable example program.
6. Describe the exceptional handling concept in detail with suitable example.
7. Explain the different types of polymorphism in detail with suitable example.
8. Discuss on constructor overloading in detail with suitable example.
9. Write a C++ program to throw multiple exceptions and define multiple catches.
10. How to overloading functions by number and type of arguments?
11. Elaborate in detail the concept of exceptional handling.

KARPAGAM ACADEMY OF HIGHER EDUCATION

(Established Under Section 3 of UGC Act 1956)

COIMBATORE – 641 021

DEPARTMENT OF COMPUTER APPLICATIONS

Batch 2016 – 2019

Subject : Programming with C and C++**Subject Code: 16MMU304B****UNIT - V**

S.No	Questions	opt1	opt2	opt3	opt4	Answer
1	In a multiple catch statement the number of throw statements are .	same as catch statement	twice than catch	only one	thrice than catch	only one
2	The exception is generated in _____ block.	try	catch	finally	throw.	try
3	The exception handling one of the function is implicitly invoked.	abort	exit	assert	throw	abort
4	The exception handling mechanism is basically built upon _____	try	catch	throw	exit	all the above
5	The point at which the throw is executed is called _____.	try	throw	throw point	exit	throw point
6	A template function may be overloaded by _____ function	template	ordinary	both (a)and (b).	virtual	both (a) and (b)
7	_____function returns true when an input or output operation has failed	eof()	fail()	bad()	good()	fail()
8	consider the following statements int x=10, y=15; x=((x<y) ? (y+x) :(y-x); what will be the value of x after executing the statements?	10	25	error.cannot be executed	5	25
9	.In multi-level inheritance, the _____ are executed in the order of	Derivations	Constructors	Destructors	Containership	constructors
10	.A class that contains objects of other classes is known as _____.	class	Nesting	Subclass	Inheritance	nesting
11	Exception handling is new feature added to -----	c++	ANSI C++	both a and b	C	ANSI C++
12	.The grand parent class is sometimes referred to as _____ class.	Ancestor	Virtual base	Indirect base	Direct base	indirect base
13	Errors such as over - flow belongs to ----- error	synchronous	Visibility	asynchronous	none of the above	synchronous
14	The errors caused by keyboards are called as----- error	asynchronous	synchronous	visibility	none of the above	asynchronous
15	The public member of a class can be accessed by its own objects using the _____ operator.	Scope resolution	Relational	Arithmetic	Dot	dot
16	The ----- should always be placed last in the list of handlers	catch(...)	catch()	throw(...)	throw()	catch(...)
17	. If the data is received from the input devices in sequence then it is called_____.	Source stream	Object stream	Destination stream	Input stream.	source stream
18	A ----- is an object that actually stores data	iterator	container	algorithm	none of the above	container
19	The _____ function takes no operator.	Operator +()	Operator –()	Friend	Conversion	operator -()
20	In overloading of binary operators, the _____ operand is used to invoke the operator function.	Right-hand	Arithmetic	Left-hand	Multiplication	left-hand
21	_____ functions may be used in place of member functions for overloading a binary operator	Inline	Member	Conversion	Friend	Friend
22	The operator that cannot be overloaded is _____	Size of	+	-	=	single of
23	The friend functions cannot be used to overload the _____ operator.	::	?:	.	=	::
24	_____ is called compile time polymorphism.	Operator overloading	Function overloading	Overloading unary operator	Overloading binary operator	operator overloading
25	_____ feature can be used to add two user-defined operator data	Function	Overloading	Arrays	Pointers	overloading
26	_____ operator cannot be overloaded.	=	+	?:	–	?:
27	Operator overloading is done with the help of a special function called _____ function.	Conversion	Operator	User-defined	In-built.	operator
28	_____ functions must either be member functions or friend functions.	Operator	User-defined	Static Member	Overloading	operator
29	The overloading operator must have atleast _____ operand that is of user-defined data type.	Two	Three	One	Four	one
30	_____ operator function should be a class member.	Arithmetic	Relational	Casting	Overloading	casting
31	The casting operator must not have any _____	Arguments	Member	Return type	Operator	arguments
32	The casting operator function must not specify a _____ type.	User-defined type	Return	Member	In-built	return
33	The operator that cannot be overloaded is _____.	Casting	Binary	Unary	Scope resolution	scope resolution
34	The friend function cannot be used to overload _____ operator.	+	-	()	::	()
35	_____ operator cannot be overloaded by friend function.	[]	*	.	?:	?:
36	The operator that cannot be overloaded by friend function is _____	.	::	->	Single of	::
37	Operator overloading is called _____	Function Overloading	Compile time polymorphism	Casting operator function	Temporary object	Compile time polymorphism
38	Overloading feature can add two _____ data types.	In-built	Enumerated	User-defined	Static	User-defined
39	The mechanism of deriving a new class from an old one is called _____	Operator overloading	Inheritance	Polymorphism	Access mechanism	polymorphism
40	_____ provides the concept of reusability.	Overloading	Message passing	Data abstraction	inheritance	inheritance
41	Only ----- operator can be overloaded.	Inheritance	Encapsulation	Polymorphism	existing	existing
42	The mechanism of deriving a class from another derived class is known as -----	Derived class	Subclass	Virtual base class	multilevel inheritance	multilevel inheritance
43	A class can inherit the attributes of two or more classes is known as -----	Abstract class	multiple inheritance	Derived class	hierarchical inheritance	multiple inheritance
44	A derived class with only one base class is called _____ inheritance.	Single	Multi-level	Multiple	Hierarchical	single
45	The derived class inherits some or all of the properties of _____	Member	Base	Father	Ancestor	base
46	A derived class can have only one _____ class.	Derived	Base	Child	Member	base
47	_____ class inherits some or all of the properties of base class.	Base	Virtual base	Subclass	Derived	derived
48	A class that inherits properties from more than one class is known as _____ inheritance.	Multiple	Multilevel	Single	Hybrid	multiple
49	A public member of a class can be accessed by its own objects using ----- operator	Hierarchical	dot (.)	Multi-level	Hybrid	dot (.)

50	The base class are separated by -----	dot (.)	commas (,)	and (&)	colon (:)	commas (,)
51	A _____ can inherit properties from more than one class.	Class	Member class		Base class	class
52	A class can be derived from another _____ class.	Member	Common base	Derived	Indirect base class	derived
53	A ----- resolution may also arise in single inheritance	ambiguity	scope	properties	Subclass	ambiguity
54	A private member of a class cannot be inherited either in public mode or in _____ mode.	Private	Protected	Visibility	Nesting	private
55	The ----- section is nothing but the body of constructor	Assignment	Private	Public	Protected	assignment
56	The initialization section basically contains a list of initializations are separated by -----	commas (,)	dot (.)	semicolon (;)	colon (:)	commas (,)
57	A _____ member of a class cannot be inherited in public mode.	Public	Protected	Private	Access	private
58	A member inherited in public mode becomes _____ in the derived	Private	Class	Public	Protected	protected
59	A protected member inherited in _____ mode becomes private in the	Protected	Visibility	Private	Public	private
60	A public member inherited in _____ mode becomes public.	Private	Public	Visibility	Protected	public
61	The mechanism of deriving certain properties of one class into another is called as -----	nesting class	member class	inheritance	Visibility	inheritance

Reg.No -----
[16MMU304B]

KARPAGAM ACADEMY OF HIGHER EDUCATION
(Established Under Section 3 of UGC Act 1956)
COIMBATORE – 641 021

B.Sc., Mathematics
(For the candidates admitted from 2016 onwards)
Third Semester
First Internal Exam July 2017
PROGRAMMING WITH C AND C++

Duration: 2 Hrs
Date & Session:

Maximum Marks: 50 Marks
Class: II BSc (Maths).,

PART-A (20 x 1 = 20 Marks)
Answer all the questions

1. C++ was originally developed by _____
a) Clocksin and Mellish b) Donald E. Knuth c) Sir Richard Hadlee d) **Bjame Strostrup**
2. C++ is an extension of C with a major addition of the class construct feature of _____
a) **Simula67** b) Simula57 c) Simula47 d) Simula87
3. C++ has the name as _____ before it was changed to C++.
a) Improved C b) Integrated C c) **C with classes** d) C with Simula
4. _____ refer to the names of variables, functions, arrays, classes etc. created by the programmer.
a) Keywords b) **Identifiers** c) Constraints d) Strings
5. The wrapping up of data and functions into a single unit is known as _____
a) abstraction b) inheritance c) polymorphism d) **encapsulation**
6. _____ refers to the act of representing essential features without including the background details or explanations.
a) **abstraction** b) inheritance c) polymorphism d) encapsulation
7. _____ is the process by which objects of one class acquire the properties of objects of another class.
a) abstraction b) **inheritance** c) polymorphism d) encapsulation
8. The _____ Operator is known as insertion operator.
a) >> b) > c) << d) <
9. In OOP, a problem is considered as a collection of number of entities called _____
a) class b) **objects** c) functions d) message
10. When a function is defined inside a class, this function is called _____
a) Inside function b) Class function c) **Inline function** d) Interior function
11. Which of the following cannot be passed to a function?
a) Reference variable b) Arrays c) Class objects d) **Header files**
12. In C++, the keyword void was used _____
a) To specify the return type of function when it is not returning any value.
b) To indicate an empty argument list to a function. c) To declare the generic pointers.
d) **All of the above.**

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

- a) procedures **b) functions** c) flowchart d) instructions

14. cout stands for

- a) class output b) character output c) common output d) call output

15. The C++ header file _____ contains function prototypes for the standard input and standard output functions.

- a) <iomanip> b) <fstream> c) <iostream> d) <cstdio>

16. The fields in a structure of a C program are by default

- a) protected b) **public** c) private d) shared

17. Everything defined at the program scope level (ie. outside functions and classes) is said to be _____

- a) local scope b) regional scope **c) global scope** d) static scope

18. Overloading is otherwise known as _____

- a) Virtual polymorphism b) transient polymorphism
c) pseudo polymorphism **d) *ad hoc* polymorphism**

19. Which of the following term is used for a function defined inside a class?

- a) member variable b) class function c) classic function d) **member function**

20. To overload an operator _____ keyword must be used along with the operator to be overloaded.

- a) Over b) Overload c) Void **d) Operator**

PART-B (3 x 2 = 6 Marks)

Answer all the questions

1. What is meant by functions?

A function is a group of statements that together perform a task. Every C++ program has at least one function, which is main(), and all the most trivial programs can define additional functions.

You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division usually is such that each function performs a specific task.

A function declaration tells the compiler about a function's name, return type, and parameters. A function definition provides the actual body of the function.

2. What is a token? What are the available tokens in C++?

A token is the smallest element of a C++ program that is meaningful to the compiler. The C++ parser recognizes these kinds of tokens: identifiers, keywords, literals, operators, punctuators, and other separators. A stream of these tokens makes up a translation unit.

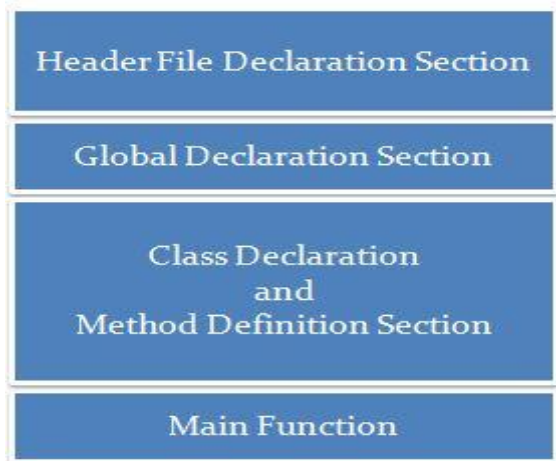
Tokens are usually separated by *white space*. White space can be one or more:

- Blanks

- Horizontal or vertical tabs
- New lines
- Formfeeds
- Comments

3. Illustrate the structure of a c++ program.

C++ Programming language is most popular language after C Programming language. C++ is first Object oriented programming language. We have summarize structure of C++ Program in the following Picture -



Section 1 : Header File Declaration Section

1. Header files used in the program are listed here.
2. Header File provides **Prototype declaration** for different library functions.
3. We can also include **user define header file**.
4. Basically all preprocessor directives are written in this section.

Section 2 : Global Declaration Section

1. Global Variables are declared here.
2. Global Declaration may include -
 - Declaring Structure
 - Declaring Class
 - Declaring Variable

Section 3 : Class Declaration Section

1. Actually this section can be considered as sub section for the global declaration section.
2. Class declaration and all methods of that class are defined here.

Section 4 : Main Function

1. Each and every C++ program always starts with main function.
2. This is entry point for all the function. Each and every method is called indirectly through main.
3. We can create class objects in the main.
4. Operating system call this function automatically.

PART-B (3 x 8= 24 Marks)

Answer all the questions

24. (a) Describe the characteristics of procedure oriented programming and object oriented programming.

Some Characteristics exhibited by procedure-oriented programming are:

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

Some of the features of object oriented programming are:

- Emphasis is on data rather than procedure.
- Programs are divided into what are known as objects.
- Data structures are designed such that they characterize the objects.
- Functions that operate on the data of an object are tied together in the data structure.
- Data is hidden and cannot be accessed by external function.
- Objects may communicate with each other through function.
- New data and functions can be easily added whenever necessary.
- Follows bottom up approach in program design.

(OR)

(b) Write a short notes on functions.

A function is a group of statements that together perform a task. Every C++ program has at least one function, which is **main()**, and all the most trivial programs can define additional functions.

A function **declaration** tells the compiler about a function's name, return type, and parameters. A function **definition** provides the actual body of the function.

The C++ standard library provides numerous built-in functions that your program can call. For example, function **strcat()** to concatenate two strings, function **memcpy()** to copy one memory location to another location and many more functions.

A function is known with various names like a method or a sub-routine or a procedure etc.

Defining a Function

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

```
return_type function_name( parameter list ) {  
    body of the function  
}
```

A C++ function definition consists of a function header and a function body. Here are all the parts of a function –

- **Return Type** – A function may return a value. The **return_type** is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword **void**.
- **Function Name** – This is the actual name of the function. The function name and the parameter list together constitute the function signature.
- **Parameters** – A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.
- **Function Body** – The function body contains a collection of statements that define what the function does.

Example

Following is the source code for a function called **max()**. This function takes two parameters num1 and num2 and return the biggest of both –

// function returning the max between two numbers

```
int max(int num1, int num2) {
    // local variable declaration
    int result;

    if (num1 > num2)
        result = num1;
    else
        result = num2;

    return result;
}
```

Function Declarations

A function **declaration** tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately.

A function declaration has the following parts –

return_type function_name(parameter list);

For the above defined function max(), following is the function declaration –

int max(int num1, int num2);

Parameter names are not important in function declaration only their type is required, so following is also valid declaration –

int max(int, int);

Function declaration is required when you define a function in one source file and you call that function in another file. In such case, you should declare the function at the top of the file calling the function.

25. (a) Explain in detail the basic concept of OOPS.

There are few principle concepts that form the foundation of object-oriented programming:

1. **Object**
2. **Class**
3. **Data Abstraction & Encapsulation**
4. **Inheritance**

5. **Polymorphism**
6. **Dynamic Binding**
7. **Message Passing**

1) **Object** :

Object is the basic unit of object-oriented programming. Objects are identified by its unique name. An object represents a particular instance of a class. There can be more than one instance of an object. Each instance of an object can hold its own relevant data.

An Object is a collection of data members and associated member functions also known as methods.

For example whenever a class name is created according to the class an object should be created without creating object can't able to use class.

The class of Dog defines all possible dogs by listing the characteristics and behaviors they can have; the object Lassie is one particular dog, with particular versions of the characteristics. A Dog has fur; Lassie has brown-and-white fur.

2) **Class**:

Classes are data types based on which objects are created. Objects with similar properties and methods are grouped together to form a Class. Thus a Class represents a set of individual objects. Characteristics of an object are represented in a class as Properties. The actions that can be performed by objects become functions of the class and is referred to as Methods.

When you define a class, you define a blueprint for an object. This doesn't actually define any data, but it does define what the class name means, that is, what an object of the class will consist of and what operations can be performed on such an object.

For example consider we have a Class of Cars under which Santro Xing, Alto and WaganR represents individual Objects. In this context each Car Object will have its own, Model, Year of Manufacture, Colour, Top Speed, Engine Power etc., which form Properties of the Car class and the associated actions i.e., object functions like Start, Move, Stop form the Methods of Car Class. No memory is allocated when a class is created. Memory is allocated only when an object is created, i.e., when an instance of a class is created.

3) **Data abstraction & Encapsulation** :

Encapsulation is placing the data and its functions into a single unit. While working with procedural languages, it is not always clear which functions work on which variables but object-oriented programming provides you framework to place the data and the relevant functions together in the same object.

When using **Data Encapsulation**, data is not accessed directly, it is only accessible through the functions present inside the class.

Data Abstraction increases the power of programming language by creating user defined data types. Data Abstraction also represents the needed information in the program without presenting the details.

Abstraction refers to the act of representing essential features without including the background details or explanation between them.

For example, a class Car would be made up of an Engine, Gearbox, Steering objects, and many more components. To build the Car class, one does not need to know how the different components work internally, but only how to interface with them, i.e., send messages to them, receive messages from them, and perhaps make the different objects composing the class interact with each other.

4) Inheritance :

One of the most useful aspects of object-oriented programming is code reusability. As the name suggests Inheritance is the process of forming a new class from an existing class or base class.

The base class is also known as parent class or super class, the new class that is formed is called derived class.

Derived class is also known as a child class or sub class. Inheritance helps in reducing the overall code size of the program, which is an important concept in object-oriented programming.

This is a very important concept of object-oriented programming since this feature helps to reduce the code size.

It is classified into different types, they are

- **Single level inheritance**
- **Multi-level inheritance**
- **Hybrid inheritance**
- **Hierarchical inheritance**

5) Polymorphism :

Polymorphism allows routines to use variables of different types at different times. An operator or function can be given different meanings or functions. Polymorphism refers to a single function or multi-functioning operator performing in different ways. Poly a Greek term means the ability to take more than one form. Overloading is one type of Polymorphism. It allows an object to have different meanings, depending on its context. When an existing operator or function begins to operate on new data type, or class, it is understood to be overloaded.

6) Dynamic binding :

Binding means connecting one program to another program that is to be executed in reply to the call. Dynamic binding is also known as late binding. The code present in the specified program is unknown till it is executed. It contains a concept of Inheritance and Polymorphism.

7) Message Passing :

An object-oriented program consists of a set of objects that communicate with each other. The process of programming in an object-oriented language, therefore, involves the following basic steps:

1. Creating classes that define objects and their behaviour
2. Creating objects from class definitions and
3. Establishing communication among objects.

Objects communicate with one another by sending and receiving information much the same way as people pass messages to one another.

A message for an object is a request for execution of a procedure, and therefore will invoke a function in the receiving object that generates the desired result . Message passing involves specifying the name of the object, the name of the function and the information to be sent.

(OR)

(b) Write a short note on built-in data types.

These data types are built-in or predefined data types and can be used directly by the user to declare variables. example: int, char , float, bool etc. Primitive data types available in C++ are:

- Integer
- Character
- Boolean
- Floating Point
- Double Floating Point
- Valueless or Void
- Wide Character

primitive data types available in C++.

- **Integer:** Keyword used for integer data types is **int**. Integers typically requires 4 bytes of memory space and ranges from -2147483648 to 2147483647.
- **Character:** Character data type is used for storing characters. Keyword used for character data type is **char**. Characters typically requires 1 byte of memory space and ranges from -128 to 127 or 0 to 255.
- **Boolean:** Boolean data type is used for storing boolean or logical values. A boolean variable can store either *true* or *false*. Keyword used for boolean data type is **bool**.
- **Floating Point:** Floating Point data type is used for storing single precision floating point values or decimal values. Keyword used for floating point data type is **float**. Float variables typically requires 4 byte of memory space.
- **Double Floating Point:** Double Floating Point data type is used for storing double precision floating point values or decimal values. Keyword used for double floating point data type is **double**. Double variables typically requires 8 byte of memory space.
- **void:** Void means without any value. void datatype represents a valueless entity. Void data type is used for those function which does not returns a value.

- **Wide Character:** Wide character data type is also a character data type but this data type has size greater than the normal 8-bit datatype. Represented by **wchar_t**. It is generally 2 or 4 bytes long.

26. (a) Write a simple program to show the use of a class.

```
// Header Files
#include <iostream>
#include<conio.h>
using namespace std;

// Class Declaration

class person {
    //Access - Specifier
public:

    //Variable Declaration
    string name;
    int number;
};

//Main Function

int main() {
    // Object Creation For Class
    person obj;

    //Get Input Values For Object Variables
    cout << "Enter the Name :";
    cin >> obj.name;

    cout << "Enter the Number :";
    cin >> obj.number;

    //Show the Output
    cout << obj.name << ": " << obj.number << endl;

    getch();
    return 0;
}
```

(OR)

(b) Write a simple program to calculate the average of two numbers.

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

int main(){
```



```
int x,y,sum;
float average;

cout << "Enter 2 integers : " << endl;
cin>>x>>y;
sum=x+y;
average=sum/2;
cout << "The sum of " << x << " and " << y << " is " << sum << "." << endl;
cout << "The average of " << x << " and " << y << " is " << average << "." << endl;
}
```

Reg.No -----

[16MMU304B]

KARPAGAM ACADEMY OF HIGHER EDUCATION
(Established Under Section 3 of UGC Act 1956)
COIMBATORE – 641 021

B.Sc., Mathematics

(For the candidates admitted from 2016 onwards)

Third Semester

First Internal Exam July 2017

PROGRAMMING WITH C AND C++

Duration: 2 Hrs

Maximum Marks: 50 Marks

Date & Session:

Class: II BSc (Maths).,

PART-A (20 x 1 = 20 Marks)

Answer all the questions

1. C++ was originally developed by _____
a) Clocksin and Mellish b) Donald E. Knuth c) Sir Richard Hadlee d) Bjarne Stroustrup
2. C++ is an extension of C with a major addition of the class construct feature of _____
a) Simula67 b) Simula57 c) Simula47 d) Simula87
3. C++ has the name as _____ before it was changed to C++.
a) Improved C b) Integrated C c) C with classes d) C with Simula
4. _____ refer to the names of variables, functions, arrays, classes etc. created by the programmer.
a) Keywords b) Identifiers c) Constraints d) Strings
5. The wrapping up of data and functions into a single unit is known as _____
a) abstraction b) inheritance c) polymorphism d) encapsulation
6. _____ refers to the act of representing essential features without including the background details or explanations.
a) abstraction b) inheritance c) polymorphism d) encapsulation
7. _____ is the process by which objects of one class acquire the properties of objects of another class.
a) abstraction b) inheritance c) polymorphism d) encapsulation
8. The _____ Operator is known as insertion operator.
a) >> b) > c) << d) <
9. In OOP, a problem is considered as a collection of number of entities called _____
a) class b) objects c) functions d) message
10. When a function is defined inside a class, this function is called _____
a) Inside function b) Class function c) Inline function d) Interior function
11. Which of the following cannot be passed to a function?
a) Reference variable b) Arrays c) Class objects d) Header files
12. In C++, the keyword void was used _____
a) To specify the return type of function when it is not returning any value.
b) To indicate an empty argument list to a function. c) To declare the generic pointers.
d) All of the above.

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

- a) procedures b) functions c) flowchart d) instructions

14. cout stands for

- a) class output b) character output c) common output d) call output

15. The C++ header file _____ contains function prototypes for the standard input and standard output functions.

- a) <iomanip> b) <fstream> c) <iostream> d) <cstdio>

16. The fields in a structure of a C program are by default

- a) protected b) public c) private d) shared

17. Everything defined at the program scope level (ie. outside functions and classes) is said to be _____

- a) local scope b) regional scope c) global scope d) static scope

18. Overloading is otherwise known as _____

- a) Virtual polymorphism b) transient polymorphism
c) pseudo polymorphism d) adhoc polymorphism

19. Which of the following term is used for a function defined inside a class?

- a) member variable b) class function c) classic function d) member function

20. To overload an operator _____ keyword must be used along with the operator to be overloaded.

- a) Over b) Overload c) Void d) Operator

PART-B (3 x 2 = 6 Marks)

Answer all the questions

1. What is meant by functions?
2. What is a token? What are the available tokens in C++?
3. Illustrate the structure of a C++ program.

PART-B (3 x 8 = 24 Marks)

Answer all the questions

24. (a) Describe the characteristics of procedure oriented programming and object oriented programming.

(OR)

(b) Write a short notes on functions.

25. (a) Explain in detail the basic concept of OOPS.

(OR)

(b) Write a short note on built-in data types.

26. (a) Write a simple program to show the use of a class.

(OR)

(b) Write a simple program to calculate the average of two numbers.

Reg.No -----
[16MMU304B]

KARPAGAM ACADEMY OF HIGHER EDUCATION

(Established Under Section 3 of UGC Act 1956)

COIMBATORE – 641 021

B.Sc., Mathematics

(For the candidates admitted from 2016 onwards)

Third Semester

Second Internal Exam August 2017

PROGRAMMING WITH C AND C++

Duration: 2 Hrs

Date & Session:

Maximum Marks: 50 Marks

Class: II BSc (Maths).,

PART-A (20 x 1 = 20 Marks)

Answer all the questions

1. C++ supports all the features of _____ as defined in C
a) structures b) union c) object d) classes
2. A structure can have both variable and functions as _____
a) objects b) classes c) members d) arguments
3. The class _____ describes the type and scope of its members
a) calling function b) declaration c) objects d) function
4. The class _____ describes how the class function are implemented
a) function definition b) declaration c) arguments d) function
5. The keywords private and public are known as _____ labels
a) static b) dynamic c) visibility d) const
6. Which is a valid method for accessing the first element of the array item?
a) item(1) b) item[1] c) item[0] d) item(0)
7. Which of the following statements is valid array declaration?
a) int number (5); b) float avg[5]; c) double [5] marks; d) counter int[5];
8. The symbol _____ is called the scope resolution operator
a) > b) :: c) << d) ::*
9. A member function can call another member function directly without using the _____ operator
a) assignment b) equal c) dot d) greater than
10. A _____ member variable is initialized to zero when the first object of its class is created
a) dynamic b) constant c) static d) protected
11. _____ Variables are normally used to maintain values common to the entire class.
a) private b) protected c) public d) static
12. When a copy of the entire object is passed to the function it is called as _____
a) pass by reference b) pass by function c) pass by pointer d) pass by value
13. When the address of the object is transferred to the function it is called as _____
a) pass by reference b) pass by function c) pass by pointer d) pass by value
14. Pointers are used to access _____

- a) Object b) Virtual function c) Class members d) functions
15. The member functions can be referred by using the _____ and _____
- a) dot operator and object b) address operator and virtual functions c) class and object
d) dot and direct operator
16. The parenthesis are necessary because the dot operator has higher precedence than the _____
- a) dot operator b) this c) class d) indirection operator
17. _____ is used to represent an object that involves a member function.
- a) friend b) this c) class d) virtual
18. The this pointer acts as an _____ argument to all the member function
- a) implicit b) explicit c) formal d) actual.
19. When two or more objects are compared inside a member function the result in return is an _____
- a) virtual function b) derived class c) invoking objects d) base class
20. Pointers are used as the objects of _____
- a) user defined b) derived class c) virtual function d) object.

PART-B (3 x 2 = 6 Marks)

Answer all the questions

21. What is an array?
22. How the objects are used as function argument?
23. Define call by reference.

PART-B (3 x 8 = 24 Marks)

Answer all the questions

24. a) Elaborate in detail the multi-dimensional array with a suitable example program.
(OR)
b) Explain on declaring and initializing unions with a simple program.
25. a) Write a note on passing entire structures to functions with a suitable example program..
(OR)
b) Write a c++ program store and calculate the sum of 5 numbers entered by the user using arrays.
26. a) Write a c++ program to convert the temperature in Celsius to Fahrenheit.
(OR)
b) Write a c++ program to check whether the number is palindrome or not.

Reg.No -----
[16MMU304B]

KARPAGAM ACADEMY OF HIGHER EDUCATION

(Established Under Section 3 of UGC Act 1956)

COIMBATORE – 641 021

B.Sc., Mathematics

(For the candidates admitted from 2016 onwards)

Third Semester

Second Internal Exam August 2017

PROGRAMMING WITH C AND C++

Duration: 2 Hrs

Date & Session:

Maximum Marks: 50 Marks

Class: II BSc (Maths).,

PART-A (20 x 1 = 20 Marks)

Answer all the questions

1. C++ supports all the features of _____ as defined in C
a) **structures** b) union c) object d) classes
2. A structure can have both variable and functions as _____
a) objects b) classes c) **members** d) arguments
3. The class _____ describes the type and scope of its members
a) calling function b) **declaration** c) objects d) function
4. The class _____ describes how the class function are implemented
a) **function definition** b) declaration c) arguments d) function
5. The keywords private and public are known as _____ labels
a) static b) dynamic c) **visibility** d) const
6. Which is a valid method for accessing the first element of the array item?
a) item(1) b) item[1] c) **item[0]** d) item(0)
7. Which of the following statements is valid array declaration?
a) int number (5); b) **float avg[5];** c) double [5] marks; d) counter int[5];
8. The symbol _____ is called the scope resolution operator
a) > b) **::** c) << d) ::*
9. A member function can call another member function directly without using the _____ operator
a) assignment b) equal c) **dot** d) greater than
10. A _____ member variable is initialized to zero when the first object of its class is created
a) dynamic b) constant c) **static** d) protected
11. _____ Variables are normally used to maintain values common to the entire class.
a) private b) protected c) public d) **static**
12. When a copy of the entire object is passed to the function it is called as _____
a) pass by reference b) pass by function c) pass by pointer d) **pass by value**
13. When the address of the object is transferred to the function it is called as _____
a) **pass by reference** b) pass by function c) pass by pointer d) pass by value
14. Pointers are used to access _____

- a) Object b) Virtual function c) **Class members** d) functions
15. The member functions can be referred by using the _____ and _____
- a) **dot operator and object** b) address operator and virtual functions c) class and object
d) dot and direct operator
16. The parenthesis are necessary because the dot operator has higher precedence than the _____
- a) dot operator b) this c) class d) **indirection operator**
17. _____ is used to represent an object that involves a member function.
- a) friend b) **this** c) class d) virtual
18. The this pointer acts as an _____ argument to all the member function
- a) **implicit** b) explicit c) formal d) actual.
19. When two or more objects are compared inside a member function the result in return is an _____
- a) virtual function b) derived class c) **invoking objects** d) base class
20. Pointers are used as the objects of _____
- a) user defined b) **derived class** c) virtual function d) object.

PART-B (3 x 2 = 6 Marks)

Answer all the questions

21. What is an array?

An **array** is a data structure which allows a collective name to be given to a group of elements which **all have the same type**. An individual element of an array is identified by its own unique **index** (or **subscript**).

An array can be thought of as a collection of numbered boxes each containing one data item. The number associated with the box is the index of the item. To access a particular item the index of the box associated with the item is used to access the appropriate box. The index **must** be an integer and indicates the position of the element in the array. Thus the elements of an array are **ordered** by the index.

22. How the objects are used as function argument?

If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the **formal parameters** of the function.

The formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.

23. Define call by reference.

This method copies the reference of an argument into the formal parameter. Inside the function, the reference is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

PART-B (3 x 8= 24 Marks)

Answer all the questions

24. a) Elaborate in detail the multi-dimensional array with a suitable example program.

C++ allows multidimensional arrays. Here is the general form of a multidimensional array declaration –

type name[size1][size2]...[sizeN];

For example, the following declaration creates a three dimensional 5 . 10 . 4 integer array –
int threedim[5][10][4];

Two-Dimensional Arrays

The simplest form of the multidimensional array is the two-dimensional array. A two-dimensional array is, in essence, a list of one-dimensional arrays. To declare a two-dimensional integer array of size x,y, you would write something as follows –

type arrayName [x][y];

Where **type** can be any valid C++ data type and **arrayName** will be a valid C++ identifier.

A two-dimensional array can be think as a table, which will have x number of rows and y number of columns. A 2-dimensional array **a**, which contains three rows and four columns can be shown as below –

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Thus, every element in array **a** is identified by an element name of the form **a[i][j]**, where **a** is the name of the array, and **i** and **j** are the subscripts that uniquely identify each element in **a**.

Initializing Two-Dimensional Arrays

Multidimensioned arrays may be initialized by specifying bracketed values for each row. Following is an array with 3 rows and each row have 4 columns.

```
int a[3][4] = {  
    {0, 1, 2, 3} , /* initializers for row indexed by 0 */  
    {4, 5, 6, 7} , /* initializers for row indexed by 1 */  
    {8, 9, 10, 11} /* initializers for row indexed by 2 */  
};
```

The nested braces, which indicate the intended row, are optional. The following initialization is equivalent to previous example –

```
int a[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
```

Accessing Two-Dimensional Array Elements

An element in 2-dimensional array is accessed by using the subscripts, i.e., row index and column index of the array. For example –

```
int val = a[2][3];
```

The above statement will take 4th element from the 3rd row of the array. You can verify it in the above digram.


```

#include <iostream>
using namespace std;

int main () {
    // an array with 5 rows and 2 columns.
    int a[5][2] = { {0,0}, {1,2}, {2,4}, {3,6},{4,8} };

    // output each array element's value
    for ( int i = 0; i < 5; i++ )
        for ( int j = 0; j < 2; j++ ) {

            cout << "a[" << i << "][" << j << "]: ";
            cout << a[i][j]<< endl;
        }

    return 0;
}

```

When the above code is compiled and executed, it produces the following result –

```

a[0][0]: 0
a[0][1]: 0
a[1][0]: 1
a[1][1]: 2
a[2][0]: 2
a[2][1]: 4
a[3][0]: 3
a[3][1]: 6
a[4][0]: 4
a[4][1]: 8

```

(OR)

b) Explain on declaring and initializing unions with a simple program.

Both structure and union are collection of different datatype. They are used to group number of variables of different type in a single unit.

Union declaration

Declaration of union must start with the keyword ***union*** followed by the union name and union's member variables are declared within braces.

Syntax for declaring union

```

union union-name
{
    datatype var1;
    datatype var2;
    -----
}

```

```
-----  
    datatype varN;  
};
```

Accessing the union members

We have to create an object of union to access its members. Object is a variable of type union. Union members are accessed using the dot operator(.) between union's object and union's member name.

Syntax for creating object of union

```
union union-name obj;
```

Example for creating object & accessing union members

```
#include<iostream.h>  
  
union Employee  
{  
    int Id;  
    char Name[25];  
    int Age;  
    long Salary;  
};  
  
void main()  
{  
  
    Employee E;  
  
    cout << "\nEnter Employee Id : ";  
    cin >> E.Id;  
  
    cout << "\nEnter Employee Name : ";  
    cin >> E.Name;  
  
    cout << "\nEnter Employee Age : ";  
    cin >> E.Age;  
  
    cout << "\nEnter Employee Salary : ";  
    cin >> E.Salary;  
  
    cout << "\n\nEmployee Id : " << E.Id;  
    cout << "\nEmployee Name : " << E.Name;  
    cout << "\nEmployee Age : " << E.Age;
```

```

        cout << "\nEmployee Salary : " << E.Salary;

    }

```

Output :

```

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

```

```

Employee Id : -20536
Employee Name : ?$?$      ?
Employee Age : -20536
Employee Salary : 45000

```

25. a) Write a note on passing entire structures to functions with a suitable example program.

When an element of a structure is passed to a function, you are actually passing the values of that element to the function. Therefore, it is just like passing a simple variable (unless, of course, that element is complex such as an array of character). For example, consider the following structure :

```

struct date
{
    short day ;
    short month ;
    short year ;
}Bdate ;

```

Individual elements of this structure can be passed as follows :
 func1(Bdate.day, Bdate.month, Bdate.year) ;

The above function-call invokes a function, func1() by passing values of individual structure elements of structure Bdate.

The function can either receive the values by creating its own copy for them (call by value) or by creating references for the original variables (call by reference). If You want that the values of the structure elements should not be altered by the function, then you should pass the structure elements by value and if you want the function to alter the original values, then you should pass the structure elements by reference.

```
/* C++ Passing Structure to Function - Call by Value */
```

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

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

```
struct distance
```

```
{
```

```
    int feet;
```

```
    int inches;
```

```
};
```

```
void prnsum(distance l1, distance l2); // function prototype
```

```
void main()
```

```
{
```

```
    clrscr();
```

```
    distance length1, length2; // two structures of type distance declared
```

```
    /* Read values for length1 */
```

```
    cout<<"Enter length 1:\n";
```

```
    cout<<"Feet: ";
```

```
    cin>>length1.feet;
```

```
    cout<<"\nInches: ";
```

```
    cin>>length1.inches;
```

```
    /* Read values for length2 */
```

```
    cout<<"\n\nEnter length 2:\n";
```

```
    cout<<"Feet: ";
```

```
    cin>>length2.feet;
```

```
    cout<<"\nInches: ";
```

```
    cin>>length2.inches;
```

```
    prnsum(length1, length2); // print sum of length1 and length2
```

```
    getch();
```

```
} // end of main()
```

```
void prnsum(distance l1, distance l2)
```

```
{
```

```
    distance l3; // new structure
```

```
    l3.feet=l1.feet+l2.feet+(l1.inches+l2.inches)/12; // 1 feet=12 inches
```

```
    l3.inches=(l1.inches+l2.inches)% 12;
```

```
    cout<<"\n\nTotal Feet: "<<l3.feet<<"\n";
```

```
    cout<<"Total Inches: "<<l3.inches;
```

```
}
```

(OR)

b) Write a c++ program store and calculate the sum of 5 numbers entered by the user using arrays.

```
include <iostream>
using namespace std;

int main()
{
    int numbers[5], sum = 0;
    cout << "Enter 5 numbers: ";

    // Storing 5 number entered by user in an array
    // Finding the sum of numbers entered
    for (int i = 0; i < 5; ++i)
    {
        cin >> numbers[i];
        sum += numbers[i];
    }

    cout << "Sum = " << sum << endl;

    return 0;
}
```

26. a) Write a c++ program to convert the temperature in Celsius to Fahrenheit.

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

void main()
{
    float cel, far;
    clrscr();
    cout<<"Enter temp. in Celsius: ";
    cin>>cel;
    far = cel * 9/5 + 32;
    cout<<"Temp. in Fahrenheit: "<<far;
    getch();
}
```

(OR)

b) Write a c++ program to check whether the number is palindrome or not.

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

int main()
{
    int n, num, digit, rev = 0;

    cout << "Enter a positive number: ";
    cin >> num;

    n = num;

    do
    {
        digit = num % 10;
        rev = (rev * 10) + digit;
        num = num / 10;
    } while (num != 0);

    cout << " The reverse of the number is: " << rev << endl;

    if (n == rev)
        cout << " The number is a palindrome";
    else
        cout << " The number is not a palindrome";

    return 0;
}
```

Reg.No -----

[16MMU304B]

Karpagam Academy of Higher Education

(Established Under Section 3 of UGC Act 1956)

COIMBATORE – 641 021

B.Sc., Mathematics

(For the candidates admitted from 2016 onwards)

Third Semester

Third Internal Exam September 2017

PROGRAMMING WITH C AND C++

Duration: 2 Hrs

Date & Session:

Maximum Marks: 50 Marks

Class: II BSc (Maths).,

PART-A (20 x 1 = 20 Marks)

Answer ALL the questions

1. _____ is used to allocate memory in the constructor
a) delete b) binding c) free d) new
2. The write() function writes _____.
a) single character b) object c) string d) signed character
3. A _____ is a collection of related data stored in a particular area on a disk.
a) field b) file c) row d) vector
4. _____ is used to free the memory
a) new b) delete c) clrscr() d) update
5. _____ inherits get(), getline(), read(), seekg(), and tellg() from istream.
a) conio b) ifstream c) fstream d) iostream
6. _____ enables an object to initialize itself when it is created
a) Destructor b) constructor c) overloading d) overriding
7. The constructor function can also be defined as _____ function
a) Friend b) inline c) default d) numeric
8. File streams act as an _____ between programs and files.
a) interface b) converter c) translator d) operator
9. Ifstream, Ofstream, Fstream are derived from _____.
a) iostream b) ostream c) streambuff d) fstreambase
10. _____ refers to the use of same thing for different purposes
a) Overloading b) Dynamic binding c) message loading d) overriding
11. _____ is to set the file buffer to read and write.
a) filebuf b) filestream c) thread d) package
12. To add data at the end of file, the file must be opened in _____ mode.
a) read() b) write() c) append() d) write and append
13. When a file is opened read or write mode a file pointer is set at _____ of the file.
a) beginning b) end c) middle d) least significant
14. The constructors that can take arguments are called _____ constructors
a) Copy b) multiple c) parameterized d) levels

15. A public member inherited in _____ mode become private in the derived class.
a) Visibility b)Private c)Protected d)Public
16. _____ are extensively used for handling class objects
a)overloaded functions b)methods c)objects d)messages
17. The eof () stands for _____.
end of file b)error opening file c) error of file d)enum of file
18. "Constructors make _____ calls to the operators new and delete when memory allocation is required"
a) Explicit b)implicit c)function d)header
19. _____ class inherits some or all of the properties of base class.
Base b)Virtual base c)Subclass d)Derived
20. Error checking does not occur during compilation if we are using_____
a) functions b)macros c)pre-defined functions d) operators

PART-B (3 x 2 = 6 Marks)

Answer ALL the questions

21. Discuss the use of malloc and calloc?
22. What is meant by free functions?
23. List out the principles of object oriented programming.

Part - C (5X6=30 Marks)

(Answer All the Questions)

24. a) Describe the concept of reading and writing text files in c++.
(OR)
b) Differentiate between static and dynamic memory allocation.
25. a) Explain in detail the preprocessor directives.
(OR)
b) Elaborate in detail the concept of multilevel inheritance with a suitable program
26. a) Write a note on template classes and their uses.
(OR)
b) Describe the exceptional handling concept in detail with suitable example.

Reg.No -----
[16MMU304B]

Karpagam Academy of Higher Education

KARPAGAM UNIVERSITY

(Established Under Section 3 of UGC Act 1956)

COIMBATORE – 641 021

B.Sc., Mathematics

(For the candidates admitted from 2016 onwards)

Third Semester

Third Internal Exam September 2017

PROGRAMMING WITH C AND C++

Duration: 2 Hrs

Date & Session:

Maximum Marks: 50 Marks

Class: II BSc (Maths).,

PART-A (20 x 1 = 20 Marks)

Answer ALL the questions

1. _____ is used to allocate memory in the constructor
a) delete b) binding c) free **d) new**
2. The write() function writes _____.
a) single character b) object c) string d) signed character
3. A _____ is a collection of related data stored in a particular area on a disk.
a) field **b) file** c) row d) vector
4. _____ is used to free the memory
a) new **b) delete** c) clrscr() d) update
5. _____ inherits get(), getline(), read(), seekg(), and tellg() from istream.
a) conio **b) ifstream** c) fstream d) iostream
6. _____ enables an object to initialize itself when it is created
a) Destructor **b) constructor** c) overloading d) overriding
7. The constructor function can also be defined as _____ function
Friend **b) inline** c) default d) numeric
8. File streams act as an _____ between programs and files.
a) interface b) converter c) translator d) operator
9. Ifstream, Ofstream, Fstream are derived form _____.
a) iostream b) ostream c) streambuff **d) fstreambase**
10. _____ refers to the use of same thing for different purposes
a) Overloading b) Dynamic binding c) message loading d) overriding
11. _____ is to set the file buffer to read and write.
a) filebuf b) filestream c) thread d) package
12. To add data at the end of file, the file must be opened in _____ mode.
a) read() b) write() **c) append()** d) write and append
13. When a file is opened read or write mode a file pointer is set at _____ of the file.
a) **beginning** b) end c) middle d) least significant
14. The constructors that can take arguments are called _____ constructors
a) Copy b) multiple **c) parameterized** d) levels

15. A public member inherited in _____ mode become private in the derived class.

- a) Visibility b) **Private** c) Protected d) Public

16. _____ are extensively used for handling class objects

- a) **overloaded functions** b) methods c) objects d) messages

17. The eof () stands for _____.

- end of file** b) error opening file c) error of file d) enum of file

18. "Constructors make _____ calls to the operators new and delete when memory allocation is required"

- a) Explicit b) **implicit** c) function d) header

19. _____ class inherits some or all of the properties of base class.

- Base b) Virtual base c) Subclass d) **Derived**

20. Error checking does not occur during compilation if we are using _____

- a) functions b) **macros** c) pre-defined functions d) operators

PART-B (3 x 2 = 6 Marks)

Answer ALL the questions

21. Discuss the use of malloc and calloc?

There are two major differences between malloc and calloc in C programming language: first, in the number of arguments. The malloc() takes a single argument, while calloc() takes two. Second, malloc() does not initialize the memory allocated, while calloc() initializes the allocated memory to ZERO.

Both malloc and calloc are used in C language for dynamic memory allocation they obtain blocks of memory dynamically. Dynamic memory allocation is a unique feature of C language that enables us to create data types and structures of any size and length suitable to our programs.

22. What is meant by free functions?

Free() Function is used to free the memory pointed by the pointer back to the memory heap. This function should be called on a pointer that was used either with "calloc()" or "malloc()", otherwise the function will destroy the memory management making a system to crash.

23. List out the principles of object oriented programming.

In order for a programming language to be object-oriented, it has to enable working with classes and objects as well as the implementation and use of the fundamental object-oriented principles and concepts: inheritance, abstraction, encapsulation and polymorphism. Let's summarize each of these fundamental principles of OOP:

- Encapsulation

It is used to hide unnecessary details in our classes and provide a clear and simple interface for working with them.

- Inheritance

It explains how class hierarchies improve code readability and enable the reuse of functionality.

- Abstraction

It helps to work through abstractions: to deal with objects considering their important characteristics and ignore all other details.

- Polymorphism

It explains how to work in the same manner with different objects, which define a specific implementation of some abstract behavior.

Part - C (5X6=30 Marks)
(Answer All the Questions)

24. a) Describe the concept of reading and writing text files in c++.

//C++ program to write and read text in/from file.

```
#include <iostream>
```

```
#include <fstream>
```

```
using namespace std;
```

```
int main()
```

```
{
```

```
    fstream file; //object of fstream class
```

```
    //opening file "sample.txt" in out(write) mode
```

```
    file.open("sample.txt",ios::out);
```

```
    if(!file)
```

```
    {
```

```
        cout<<"Error in creating file!!!"<<endl;
```

```
        return 0;
```

```
    }
```

```
    cout<<"File created successfully."<<endl;
```

```
    //write text into file
```

```
    file<<"ABCD.";
```

```
    //closing the file
```

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

```
    //again open file in read mode
```

```
    file.open("sample.txt",ios::in);
```

```
    if(!file)
```

```
    {
```

```
        cout<<"Error in opening file!!!"<<endl;
```

```
        return 0;
```

```
    }
```

```
    //read untill end of file is not found.
```

```

char ch; //to read single character
cout<<"File content: ";

while(!file.eof())
{
    file>>ch; //read single character from file
    cout<<ch;
}

file.close(); //close file

return 0;
}

```

(OR)

b) Differentiate between static and dynamic memory allocation.

Static memory allocation: The compiler allocates the required memory space for a declared variable. By using the address of operator, the reserved address is obtained and this address may be assigned to a pointer variable. Since most of the declared variables have static memory, this way of assigning pointer value to a pointer variable is known as static memory allocation. Memory is assigned during compilation time.

Dynamic memory allocation: It uses functions such as `malloc()` or `calloc()` to get memory dynamically. If these functions are used to get memory dynamically and the values returned by these functions are assigned to pointer variables, such assignments are known as dynamic memory allocation. Memory is assigned during run time.

25. a) Explain in detail the preprocessor directives.

Preprocessor directives are lines included in the code of programs preceded by a hash sign (#). These lines are not program statements but directives for the *preprocessor*. The preprocessor examines the code before actual compilation of code begins and resolves all these directives before any code is actually generated by regular statements.

These *preprocessor directives* extend only across a single line of code. As soon as a newline character is found, the preprocessor directive ends. No semicolon (;) is expected at the end of a preprocessor directive. The only way a preprocessor directive can extend through more than one line is by preceding the newline character at the end of the line by a backslash (\).

macro definitions (#define, #undef)

To define preprocessor macros we can use `#define`. Its syntax is:

```

#define          identifier          replacement

```

When the preprocessor encounters this directive, it replaces any occurrence of identifier in the rest of the code by replacement. This replacement can be an expression, a statement, a block or simply anything. The preprocessor does not understand C++ properly, it simply replaces any occurrence of identifier by replacement.

```

1 #define TABLE_SIZE 100
2 int table1[TABLE_SIZE];
3 int table2[TABLE_SIZE];

```

After the preprocessor has replaced `TABLE_SIZE`, the code becomes equivalent to:

```
1 int table1[100];
2 int table2[100];
```

`#define` can work also with parameters to define function macros:

```
#define getmax(a,b) a>b?a:b
```

This would replace any occurrence of `getmax` followed by two arguments by the replacement expression, but also replacing each argument by its identifier,

```
1 //function macro
2 #include <iostream>
3 using namespace std;
4
5 #define getmax(a,b) ((a)>(b)?(a):(b))
6
7 int main()
8 {
9     int x=5, y;
10    y= getmax(x,2);
11    cout << y << endl;
12    cout << getmax(7,x) << endl;
13    return 0;
14 }
```

5 [Edit & Run](#)
7

Defined macros are not affected by block structure. A macro lasts until it is undefined with the `#undef` preprocessor directive:

```
1 #define TABLE_SIZE 100
2 int table1[TABLE_SIZE];
3 #undef TABLE_SIZE
4 #define TABLE_SIZE 200
5 int table2[TABLE_SIZE];
```

This would generate the same code as:

```
1 int table1[100];
2 int table2[200];
```

Function macro definitions accept two special operators (`#` and `##`) in the replacement sequence:

The operator `#`, followed by a parameter name, is replaced by a string literal that contains the argument passed (as if enclosed between double quotes):

```
1 #define str(x) #x
2 cout << str(test);
```

This would be translated into:

```
cout << "test";
```

The operator `##` concatenates two arguments leaving no blank spaces between them:

```
1 #define glue(a,b) a ## b
2 glue(c,out) << "test";
```

This would also be translated into:

```
cout << "test";
```

Because preprocessor replacements happen before any C++ syntax check, macro definitions can be a tricky feature. But, be careful: code that relies heavily on complicated macros become less readable, since the syntax expected is on many occasions different from the normal expressions programmers expect in C++.

(OR)

b) Elaborate in detail the concept of multilevel inheritance with a suitable program

Inheritance is one of the core feature of an object-oriented programming language. It allows software developers to derive a new class from the existing class. The derived class inherits the features of the base class (existing class).

In C++ programming, not only you can derive a class from the base class but you can also derive a class from the derived class. This form of inheritance is known as multilevel inheritance.

```
class A
{
... ..
};
class B: public A
{
... ..
};
class C: public B
{
... ..
};
```

Here, class *B* is derived from the base class *A* and the class *C* is derived from the derived class *B*.

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

```

class A
{
    public:
        void display()
        {
            cout<<"Base class content.";
        }
};

class B : public A
{

};

class C : public B
{

};

int main()
{
    C obj;
    obj.display();
    return 0;
}

```

26. a) Write a note on template classes and their uses.

Templates are a way of making classes more abstract by letting to define the behavior of the class without actually knowing what datatype will be handled by the operations of the class. In essence, this is what is known as generic programming; this term is a useful way to think about templates because it helps remind the programmer that a templated class does not depend on the datatype (or types) it deals with. To a large degree, a templated class is more focused on the algorithmic thought rather than the specific nuances of a single datatype. Templates can be used in conjunction with abstract datatypes in order to allow them to handle any type of data.

The format for declaring function templates with type parameters is:

```

template <class identifier> function_declaration;
template <typename identifier> function_declaration;

```

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

One example is:

```

template <class myType>
myType GetMax (myType a, myType b) {

```

```
return (a>b?a:b);  
}
```

(OR).

b) Describe the exceptional handling concept in detail with suitable example.

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:

```
#include <iostream>
```



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

struct MyException : public exception {
    const char * what () const throw () {
        return "C++ Exception";
    }
};

int main() {
    try {
        throw MyException();
    } catch(MyException& e) {
        std::cout << "MyException caught" << std::endl;
        std::cout << e.what() << std::endl;
    } catch(std::exception& e) {
        //Other errors
    }
}
```