KARPAGAM ACADEMY OF HIGHER EDUCATION

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

Coimbatore - 641021.

(For the candidates admitted from 2017 onwards)

DEPARTMENT OF COMPUTER SCIENCE,CA & IT

SUBJECT : **PROGRAMMING FUNDAMENTALS USING C/C++**

SEMESTER : **I**   **L T P C**

SUBJECT CODE: **17CSU101**   **CLASS** : **I B.Sc.CS A**   **4 0 0 4**

Course Objective:

1. To learn advanced features of the C++ programming language as a continuation of the previous course.

2. To learn the characteristics of the object-oriented programming language: data abstraction and information hiding, inheritance, and dynamic binding of the messages to the methods.

3. To learn the basic principles of the object-oriented design and software engineering in terms of software reuse and managing the complexity.

4. To enhance problem solving and programming skills in C++ with extensive programming projects.

5. To become familiar with the UNIX software development environment.

Course Learning Outcomes:

After completion of this course, a student will know to do the following

1. To use the characteristics of the object-oriented programming language in a program.

2. To use the basic object-oriented design principles in computer problem solving.

3. To use the basic principles of software engineering in managing complex software projects.

4. To write program with the advanced features of the C++ programming language.

5. To develop programs in the UNIX programming environment.

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, putcharetc), Formatted and Console I/O (printf(), scanf(), cin, cout), Using Basic Header Files (stdio.h, iostream.h, conio.hetc).**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 Access, 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. Herbtz Schildt, 2003,*C++: The Complete Reference*, 4th Edition, McGraw Hill.

2. Bjarne Stroustrup, 2013, *The C++ Programming Language,* 4th Edition, Addison-Wesley.

3. Bjarne Stroustroup, 2014, *Programming - Principles and Practice using C++,* 2nd Edition, Addison-Wesley.

4. E Balaguruswamy, 2008,*Object Oriented Programming with C++,* 2nd Edition ,Tata McGraw-Hill Education.

5. Paul Deitel, Harvey Deitel, 2011,*C++ How to Program*, 8th Edition, Prentice Hall.

6. John R. Hubbard, 2000, *Programming with C++,* 2nd Edition ,Schaum's Series.

7. Stefan Bjornander, 2016,*C++ Windows Programming,* Published by Packt Publishing Ltd.

8. Scott Meyers, 2005, *Effective C++,* 3rd Edition, Published by Addison-Wesley.

9. Harry, H. Chaudhary, 2014 ,*Head First C++ Programming: The Definitive Beginner's Guide*, First Create space Inc, 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*, 5th Edition, Published by Addison-Wesley.

12. Debasish Jana , 2014,*C++ And Object-Oriented Programming Paradigm*,Published by PHI Learning Pvt. Ltd.

13. Richard L. Stegman, 2016, *Focus on Object-oriented Programming With C++*,6th Edition ,CreateSpace Independent Publishing Platform,.

14. Andrew Koeni, Barbara, E. Moo,2000,*Accelerated using C++,* Published by Addison-Wesley .

WEBSITES

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. http://www.cplusplus.com/
6. http://www.cppreference.com/

KARPAGAM ACADEMY OF HIGHER EDUCATION

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

Coimbatore - 641021.

(For the candidates admitted from 2017 onwards)

DEPARTMENT OF COMPUTER SCIENCE,CA & IT

SUBJECT   : **PROGRAMMING FUNDAMENTALS USING C/C++**

SEMESTER   : **I**   **L T P C**

SUBJECT CODE: **17CSU101**   **CLASS**   : **I B.Sc.CS**   **4 0 0 4**

LECTURE PLAN
STAFF NAME: Dr.P.TAMIL SELVAN, S.A. SATHYA PRABHA

S.No	Lecture Duration (Hr)	Topics	Support Materials
		UNIT-I	
1.	1	**Introduction to C and C++:** ➢ History of C and C++, Overview of Procedural Programming and Object-Orientation Programming	T1:1-3, 12-14 T2:4-7
2.	1	➢ Using main() function, Compiling and Executing Simple Programs in C++.	T1:12-14 T2:28
3.	1	➢ **Data Types, Variables, Constants, Operators and Basic I/O:** Declaration, Defining and Initializing Variables, Scope of Variables	T1:30-31, 34-35 T2:42-46
4.	1	➢ Using Named Constants, Keywords	T1:25-30,31-34
5.	1	➢ Data Types, Casting of Data Types	T2:32-37
6.	1	➢ Operators, Using Comments in programs	T1:52-61 T2: 46-49
7.	1	➢ Character I/O, Formatted and Console I/O, Using Basic Header Files	T1:84-98 T2:21, 248-266
8.	1	➢ **Expressions, Conditional Statements and Iterative Statements:** Simple Expressions in C++	T1: 63-67
9.	1	➢ Understanding Operators Precedence in Expressions	T2: 54-56

10.	1	➢ Conditional Statements, Understanding syntax and utility of Iterative Statements, , Use of break and continue in Loops, Using Nested Statements	T1:114-126,
11.	1	➢ Use of break and continue in Loops, Using Nested Statements, understanding syntax and utility of Iterative Statements	T1: 152-166
12.	1	Recapitulation and Discussion of important questions	
Text Book	**T1:** BjarneStroustrup, 2014, *Programming - Principles and Practice using C++,* 2nd Edition, Addison- Wesley. **T2:** Bjarne Stroustrup, 2013, *The C++ Programming Language,* 4th Edition. Addison-Wesley.		
		Total No of Hours Planned For Unit – I	**12**
		UNIT-II	
1.	1	**Functions and Arrays**: ➢ Utility of functions, Call by Value, Call by Reference	T1:270-272
2.	1	➢ Functions returning value	T1: 269-272
3.	1	➢ Void functions	T1: 274
4.	1	➢ Inline Functions	T2: 75-77
5.	1	➢ Return data type of functions, Functions parameters, Differentiating between Declaration and Definition of Functions,	T1:272-274
6.	1	➢ Command Line Arguments/Parameters in Functions,	T1:405-408 T2:301-303
7.	1	➢ Functions with variable number of Arguments.	T1:285-286
8.	1	➢ Creating and Using One Dimensional Arrays	T1: 192-199
9.	1	➢ Various types of arrays	T1: 209-210
10.	1	➢ Two-dimensional Arrays, Introduction to Multi-dimensional arrays.	T1:199-209
11.	1	Recapitulation and Discussion of important questions	
Text Book	**T1:** BjarneStroustrup, 2014, *Programming - Principles and Practice using C++,* 2nd Edition, Addison- Wesley. **T2:** Bjarne Stroustrup, 2013, *The C++ Programming Language,* 4th Edition. Addison-Wesley.		

			Total No of Hours Planned For Unit – II	11
			UNIT-III	
1.		1	**Derived Data Types (Structures and Unions):** ➢ Understanding utility of structures and unions	T1: 371-319,
2.		1	➢ Declaring, initializing and using simple structures and unions	T1: 322-324
3.		1	➢ Manipulating individual members of structures and unions	T1: 321-322,
4.		1	➢ Array of Structures, Individual data members as structures	T1: 326-329
5.		1	➢ Passing and returning structures from functions.	T1: 333-335
6.		1	➢ Structure with Union as members, Union with Structures as members	T1: 335-337
7.		1	**Pointers and References in C++:** ➢ Understanding a Pointer Variable, Simple use of Pointers	T1:351-355
8.		1	➢ Pointers to Pointers, Pointers to Structures	T1: 376-379
9.		1	➢ Passing pointers as function arguments, Returning a pointer from a function	T1:370-373
10.		1	➢ Using arrays as pointers, Passing arrays to functions.	T1:369-370
11.		1	Recapitulation and Discussion of important questions	
Text Book		**T1:** BjarneStroustroup, 2014, *Programming - Principles and Practice using C++,* 2nd Edition, Addison- Wesley.		
			Total No of Hours Planned For Unit – III	11
			UNIT-IV	
1.		1	**File I/O, Preprocessor Directives:** ➢ Opening and closing a file	T1:389-392
2.		1	➢ Reading and writing Text Files	T1:392-394
3.		1	➢ Using put() statement	T1:394-396
4.		1	➢ Using get() statement	T1:396-398
5.		1	➢ Using read() functions	T1:289-291
6.		1	➢ Using write() functions	T1:291-294
7.		1	➢ Random access in files	T1:400-405 T2:294-299
8.		1	➢ Understanding the Preprocessor Directives	T2:444-445, 449-453

9.	1	➢ Macros.	T2:445-449
10.	1	Recapitulation and Discussion of important questions	
Text Book	**T1:** BjarneStroustroup, 2014, *Programming - Principles and Practice using C++,* 2nd Edition, Addison- Wesley. **T2:** Bjarne Stroustrup, 2013, *The C++ Programming Language,* 4th Edition. Addison-Wesley.		
		Total No of Hours Planned For Unit – IV	**10**
		UNIT-V	
1.	1	**Using Classes in C++:** ➢ Principles of Object-Oriented Programming,	T2: 88-90
2.	1	➢ Defining & Using Classes	T2: 91-96
3.	1	➢ Constructors	T2:127-133
4.	1	➢ Constructor Overloading	T2:130-133
5.	1	➢ Function overloading	T2:80-82,
6.	1	➢ Operator overloading	T2:150-157
7.	1	➢ Class Variables & Functions, access specifiers	T2:98,107-109
8.	1	➢ Overview of Template classes and their use	T2:308-314
9.	1	➢ Inheritance, Polymorphism and Exception Handling	T2:176-179
10.	1	➢ Introduction to Inheritance	T2:180-182
11.	1	➢ Polymorphism	T2: 222-223
12.	1	➢ Basics Exceptional Handling	T2:326-332
13.	1	Recapitulation and Discussion of important questions	
14.	1	**Discussion of previous ESE Question papers**	
15.	1	**Discussion of previous ESE Question papers**	
16.	1	**Discussion of previous ESE Question papers**	
Text Book	**T2:** Bjarne Stroustrup, 2013, *The C++ Programming Language,* 4th Edition. Addison-Wesley.		
		Total No of Hours Planned For Unit – V	**16**
		Total No. of Hours Planned: 60	

TEXT BOOKS

T1: Bjarne Stroustroup, 2014, *Programming - Principles and Practice using C++,* 2nd Edition ,Addison- Wesley.

T2: Bjarne Stroustrup, 2013, *The C++ Programming Language,* 4th Edition, Addison-Wesley.

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

T4: Stanley B. Lippman, Josee Lajoie, Barbara E. Moo, 2012, *C++ Primer*, 5thEdition ,Published by Addison-Wesley.

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

T6: E Balaguruswamy, 2008*,Object Oriented Programming with C++,* 2nd Edition ,Tata McGraw-Hill Education.

T7: Walter Savitch, 2007*,Problem Solving with C++,* Pearson Education.

T8: Scott Meyers, 2005, *Effective C++,*3rd Edition ,Published by Addison-Wesley.

T9 : Debasish Jana , 2014*,C++ And Object-Oriented Programming Paradigm*,Published by PHI Learning Pvt. Ltd.

T10: Richard L. Stegman, 2016, *Focus on Object-oriented Programming With C++*, 6th Edition. CreateSpace Independent Publishing Platform.

T11. Andrew Koeni, Barbara, E. Moo, 2000, Accelerated *using C++,* Published by Addison-Wesley.

WEBSITES

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. http*://*www.cplusplus.com/

KARPAGAM ACADEMY OF HIGHER EDUCATION

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

Coimbatore - 641021.

(For the candidates admitted from 2017 onwards)

DEPARTMENT OF COMPUTER SCIENCE,CA & IT

SUBJECT : **PROGRAMMING FUNDAMENTALS USING C/C++**

SEMESTER : **I**   **L T P C**

SUBJECT CODE: **17CSU101**   **CLASS** : **I B.Sc.CS**   **4 0 0 4**

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, putcharetc), Formatted and Console I/O (printf(), scanf(), cin, cout), Using Basic Header Files (stdio.h, iostream.h, conio.hetc).**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).

Suggested Readings:

1. Herbtz Schildt, 2003,*C++: The Complete Reference*, 4th Edition, McGraw Hill.

2. Bjarne Stroustrup, 2013, *The C++ Programming Language,* 4th Edition, Addison-Wesley.

3. Bjarne Stroustroup, 2014, *Programming - Principles and Practice using C++,* 2nd Edition, Addison-Wesley.

4. E Balaguruswamy, 2008,*Object Oriented Programming with C++,* 2nd Edition ,Tata McGraw-Hill Education.

5. Paul Deitel, Harvey Deitel, 2011,*C++ How to Program*, 8th Edition, Prentice Hall.

6. John R. Hubbard, 2000, *Programming with C++,* 2nd Edition ,Schaum's Series.

7. Stefan Bjornander, 2016,*C++ Windows Programming,* Published by Packt Publishing Ltd.

8. Scott Meyers, 2005, *Effective C++,* 3rd Edition, Published by Addison-Wesley.

9. Harry, H. Chaudhary, 2014 ,*Head First C++ Programming: The Definitive Beginner's Guide*, First Create space Inc, 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*, 5th Edition, Published by Addison-Wesley.

12. Debasish Jana , 2014,*C++ And Object-Oriented Programming Paradigm*,Published by PHI Learning Pvt. Ltd.

13. Richard L. Stegman, 2016, *Focus on Object-oriented Programming With C++,*6th Edition ,CreateSpace Independent Publishing Platform,.

14. Andrew Koeni, Barbara, E. Moo,2000,*Accelerated using C++,* Published by Addison-Wesley .

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. http://www.cplusplus.com/

6. http://www.cppreference.com/

UNIT 1

Introduction to computers
Computer:

It is an electronic device, It has memory and it performs arithmetic and logical operations.

Input:

The data entering into computer is known as input.

Output:

The resultant information obtained by the computer is known as output.

Program:

A sequence of instructions that can be executed by the computer to solve the given problem is known as program.

Software:

A set of programs to operate and controls the operation of the computer is known as software.

these are 2 types.

1.System software.

2.Application software.

System Software:

It is used to manages system resources.

Eg: Operating System.

Operating system:

It is an interface between user and the computer. In other words operating system is a complex set of programs which manages the resources of a computer. Resources include input, output, processor,memory,etc. So it is called as Resource Manager.

Eg: Windows 98,WindowsXp,Windows7,Unix, Linux ,etc.

Application Software:

It is Used to develop the applications.

It is again of 2 types.

1.Languages

2.Packages.

Language:

It consists a set of executable instructions. Using these instructions we can communicate with the computer and get the required results.

Eg: C, C++,Java, etc.

Hardware:

All the physical components or units which are connecting to the computer circuit is known as Hardware.

ASCII character Set

ASCII - American Standard Code for Information Interchange

There are 256 distinct ASCII characters are used by the micro computers. These values range from 0 to 255. These can be grouped as follows.

Character Type	No. of Characters
Capital Letters (A to Z)	26
Small Letters (a to z)	26
Digits (0 to 9)	10
Special Characters	32
Control Characters	34
Graphic Characters	128
Total	256

Out of 256, the first 128 are called as ASCII character set and the next 128 are called as extended ASCII character set. Each and every character has unique appearance.

Eg:

A to Z	65 to 90
a to z	97 to 122
0 to 9	48 to 57
Esc	27

Backspace	8
Enter	13
SpaceBar	32
Tab	9

Classification of programming languages:-

Programming languages are classifies into 2 types

1.Low level languages

2.High level languages

Low level languages:

It is also known as Assembly language and was designed in the beginning. It has some simple instructions. These instructions are not binary codes, but the computer can understand only the machine language, which is in binary format. Hence a converter or translator is used to translate the low level language instructions into machine language. This translator is called as assembler.

High level languages:

These are more English like languages and hence the programmers found them very easy to learn. To convert high level language instructions into machine language compilers and interpreters are used.

Translators:

These are used to convert low or high level language instructions into machine language with the help of ASCII character set. There are 3 types of translators for languages.

1) **Assembler :**

It is used to convert low level language instructions into machine language.

2) **Compiler:**

It is used to convert high level language instructions into machine language. It checks for the errors in the entire program and converts the program into machine language.

3) **Interpreter:**

It is also used to convert high level language instructions into machine language, But It checks for errors by statement wise and converts into machine language.

Debugging :

The process of correcting errors in the program is called as debugging.

Introduction to C and C++:

C is computer programming language. It was designed by Dennis Ritchie at AT &T (American Telephones and Telegraphs) BELL labs in USA.

It is the most popular general purpose programming language. We can use the 'C' language to implement any type of applications. Mainly we are using C language to implement system software. These are compilers, editors, drivers, databases and operating systems.

A Brief History of C:

The C programming language was developed at Bell Labs during the early 1970's. Quite unpredictably it derived from a computer language named B and from an earlier language BCPL. Initially designed as a system programming language under UNIX it expanded to have wide usage on many different systems. The earlier versions of C became known as K&R C after the authors of an earlier book, "The C Programming Language" by Kernighan and Ritchie. As the language further developed and standardized, a version know as ANSI (American National Standards Institute) C became dominant. As you study this language expect to see references to both K&R and ANSI C. Although it is no longer the language of choice for most new development, it still is used for some system and network programming as well as for embedded systems. More importantly, there is still a tremendous amount of legacy software still coded in this language and this software is still actively maintained.

A Brief History of C++:

Bjarne Stroustrup at Bell Labs initially developed C++ during the early 1980's. It was designed to support the features of C such as efficiency and low-level support for system level coding. Added to this were features such as classes with inheritance and virtual functions, derived from the Simula67 language, and operator overloading, derived from Algol68. Don't worry about understanding all the terms just yet, they are explained in easyCPlusPlus's C++ tutorials. C++ is best described as a superset of C, with full support for object-oriented programming. This language is in wide spread use.

Differences between C and C++:

Although the languages share common syntax they are very different in nature. C is a procedural language. When approaching a programming challenge the general method of solution is to break the task into successively smaller subtasks. This is known as top-down design. C++ is an

object-oriented language. To solve a problem with C++ the first step is to design classes that are abstractions of physical objects. These classes contain both the state of the object, its members, and the capabilities of the object, its methods. After the classes are designed, a program is written that uses these classes to solve the task at hand.

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

	Procedure Oriented Programming	**Object Oriented Programming**
Divided Into	In POP, program is divided into small parts called functions.	In OOP, program is divided into parts called objects.
Importance	In POP, Importance is not given to data but to functions as well as sequence of actions to be done.	In OOP, Importance is given to the data rather than procedures or functions because it works as a real world.
Approach	POP follows Top Down approach.	OOP follows Bottom Up approach.
Access Specifiers	POP does not have any access specifier.	OOP has access specifiers named Public, Private, Protected, etc.
Data Moving	In POP, Data can move freely from function to function in the system.	In OOP, objects can move and communicate with each other through member functions.
Expansion	To add new data and function in POP is not so easy.	OOP provides an easy way to add new data and function.

Data Access	In POP, Most function uses Global data for sharing that can be accessed freely from function to function in the system.	In OOP, data cannot move easily from function to function, it can be kept public or private so we can control the access of data.
Data Hiding	POP does not have any proper way for hiding data so it is **less secure**.	OOP provides Data Hiding so provides more security.
Overloading	In POP, Overloading is not possible.	In OOP, overloading is possible in the form of Function Overloading and Operator Overloading.
Examples	Example of POP is: C, VB, FORTRAN, and Pascal.	Example of OOP is: C++, JAVA, VB.NET, C#.NET.

Main Function:

A program shall contain a global function named **main**, which is the designated start of the program.

int **main** () { *body* }
int **main** (int *argc*, char **argv*[] , *other_parameters*) { *body* }

argc - Non-negative value representing the number of arguments passed to the program from the environment in which the program is run.

argv - Pointer to the first element of an array of pointers to null-terminated multibyte strings that represent the arguments passed to the program from the execution environment (argv[0]through argv[argc-1]). The value of argv[argc] is guaranteed to be 0.

body   -   The body of the main function

other_parameters   -   Implementations may allow additional forms of the main function as long as the return type remains int. A very common extension is passing a third argument of type char*[] pointing at an array of pointers to the execution environment variables.

The names argc and argv are arbitrary, as well as the representation of the types of the parameters: int main(int ac, char** av) is equally valid.

Explanation:

The main function is called at program startup after initialization of the non-local objects with static storage duration. It is the designated entry point to a program that is executed in *hosted* environment   (that   is,   with   an   operating   system).   The   entry   points to *freestanding* programs (boot loaders, OS kernels, etc) are implementation-defined.

The main function has several special properties:

1) It cannot be used anywhere in the program

a) in particular, it cannot be called recursively

b) its address cannot be taken

2) It cannot be predefined and cannot be overloaded: effectively, the name main in the global namespace is reserved for functions (although it can be used to name classes, namespaces, enumerations, and any entity in a non-global namespace, except that a function called "main" cannot be declared with C language linkage in any namespace.

3) It cannot be defined as deleted or declared with C language linkage, inline, static, or constexpr

4) The body of the main function does not need to contain the return statement: if control reaches the end of main without encountering a return statement, the effect is that of executing return 0;

5) Execution of the return (or the implicit return upon reaching the end of main) is equivalent to first leaving the function normally (which destroys the objects with automatic storage duration)

and then calling std::exitwith the same argument as the argument of the return. (std::exit then destroys static objects and terminates the program)

6) If the main function is defined with a function-try-block, the exceptions thrown by the destructors of static objects (which are destroyed by the implied std::exit) are not caught by it.

7) The return type of the main function cannot be deduced (auto main() {... is not allowed)

C/C++ Program Compilation:

Creating, Compiling and Running Your Program:
The stages of developing your C program are as follows.

Creating the program:
Create a file containing the complete program, such as the above example. You can use any Ordinary editor with which you are familiar to create the file. One such editor is *textedit* available on most UNIX systems.

The filename must by convention end ``.c'' (full stop, lower case c), *e.g. myprog.c* or *progtest.c*. The contents must obey C syntax. For example, they might be as in the above example, starting with the line /* Sample.... (or a blank line preceding it) and ending with the line } /* end of program */ (or a blank line following it).

Compilation:
There are many C compilers around. The cc being the default Sun compiler. The GNU C compiler gcc is popular and available for many platforms. PC users may also be familiar with the Borland bcc compiler.

There are also equivalent C++ compilers which are usually denoted by CC (upper case CC. For example Sun provides CC and GNU GCC. The GNU compiler is also denoted by g++

Other (less common) C/C++ compilers exist.

All the above compilers operate in essentially the same manner and share many common command line options.

To compile your program simply invoke the command cc. The command must be followed by the name of the (C) program you wish to compile. A number of compiler options can be specified also.

Thus, the basic compilation command is:

cc program.c

Where *program.c* is the name of the file.

If there are obvious errors in your program (such as mistyping, misspelling one of the key words or omitting a semi-colon), the compiler will detect and report them.

There may, of course, still be logical errors that the compiler cannot detect. You may be telling the computer to do the wrong operations.

When the compiler has successfully digested your program, the compiled version, or executable, is left in a file called *a.out* or if the compiler option *-o* is used : the file listed after the *-o*.

It is more convenient to use a *-o* and filename in the compilation as in

cc -o program program.c

Which puts the compiled program into the file program (or any file you name following the "-o" argument) **instead** of putting it in the file a.out.

Running the program:

The next stage is to actually run your executable program. To run an executable in UNIX, you simply type the name of the file containing it, in this case *program* (or *a.out*)

This executes your program, printing any results to the screen. At this stage there may be run-time errors, such as division by zero, or it may become evident that the program has produced incorrect output.

If so, you must return to edit your program source, and recompile it, and run it again.

The C Compilation Model:
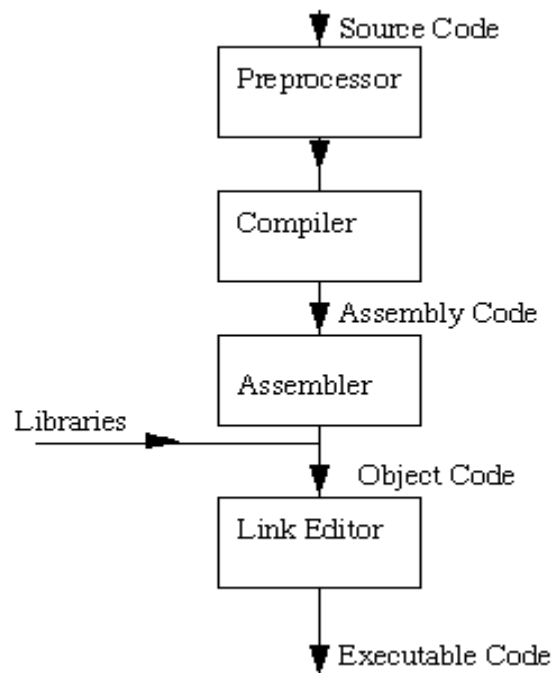
The C Compilation Model:

The Preprocessor

We will study this part of the compilation process in greater detail later

The Preprocessor accepts source code as input and is responsible for

- removing comments

- Interpreting special *preprocessor directives* denoted by #.

For example

- #include -- includes contents of a named file. Files usually called *header* files. *e.g*

    o   #include <math.h> -- standard library maths file.

    o   #include <stdio.h> -- standard library I/O file

- #define -- defines a symbolic name or constant. Macro substitution.

    o   #define MAX_ARRAY_SIZE 100

C Compiler:

The C compiler translates source to assembly code. The source code is received from the preprocessor.

Assembler:

The assembler creates object code. On a UNIX system you may see files with a .o suffix (.OBJ on MSDOS) to indicate object code files.

Link Editor:

If a source file references library functions or functions defined in other source files the *link editor* combines these functions (with main()) to create an executable file. External Variable references resolved here also.

Primitive Built-in Data Types:

C++ offer the programmer a rich assortment of built-in as well as user defined data types. Following table lists down seven basic C++ data types:

Type	Keyword
Boolean	bool
Character	char
Integer	int
Floating point	float
Double floating point	double
Valueless	void
Wide character	wchar_t

Several of the basic types can be modified using one or more of these type modifiers:

- signed
- unsigned
- short
- long

The following table shows the variable type, how much memory it takes to store the value in memory, and what is maximum and minimum value which can be stored in such type of variables.

Type	Typical Bit Width	Typical Range
char	1byte	-128 to 127 or 0 to 255
unsigned char	1byte	0 to 255
signed char	1byte	-128 to 127
int	4bytes	-2147483648 to 2147483647
unsigned int	4bytes	0 to 4294967295
signed int	4bytes	-2147483648 to 2147483647
short int	2bytes	-32768 to 32767
unsigned short int	2bytes	0 to 65,535
signed short int	2bytes	-32768 to 32767
long int	4bytes	-2,147,483,648 to 2,147,483,647
signed long int	4bytes	-2,147,483,648 to 2,147,483,647
unsigned long int	4bytes	0 to 4,294,967,295
float	4bytes	+/- 3.4e +/- 38 (~7 digits)
double	8bytes	+/- 1.7e +/- 308 (~15 digits)
long double	8bytes	+/- 1.7e +/- 308 (~15 digits)

wchar_t	2 or 4 bytes	1 wide character

The sizes of variables might be different from those shown in the above table, depending on the compiler and the computer you are using.

Following is the example, which will produce correct size of various data types on your computer.

```
#include <iostream.h>
int main()
{
cout <<"Size of char : "<< sizeof(char) << endl;
cout <<"Size of int : "<< sizeof(int) << endl;
cout <<"Size of short int : "<< sizeof(short int) << endl;
cout <<"Size of long int : "<< sizeof(long int) << endl;
cout <<"Size of float : "<< sizeof(float) << endl;
cout <<"Size of double : "<< sizeof(double) << endl;
cout <<"Size of wchar_t : "<< sizeof(wchar_t) << endl;
return 0;
}
```

This example uses **endl**, which inserts a new-line character after every line and << operator is being used to pass multiple values out to the screen. We are also using **sizeof**() operator to get size of various data types.

When the above code is compiled and executed, it produces the following result which can vary from machine to machine:

Size of char : 1

Size of int : 4

Size of short int : 2

Size of long int : 8

Size of float : 4

Size of double : 8

Size of wchar_t : 4

Variables:

While doing programming in any programming language, you need to use various variables to store various information. Variables are nothing but reserved memory locations to store values. This means that when you create a variable you reserve some space in memory. You may like to store information of various data types like character, wide character, integer, floating point, double floating point, Boolean etc. Based on the data type of a variable, the operating system allocates memory and decides what can be stored in the reserved memory.

What are Variables?

Variable are used in C++, where we need storage for any value, which will change in program. Variable can be declared in multiple ways each with different memory requirements and functioning. Variable is the name of memory location allocated by the compiler depending upon the datatype of the variable.

Example : `int i=10;` // declared and initialised

Memory Location reserved and is named as i

RAM
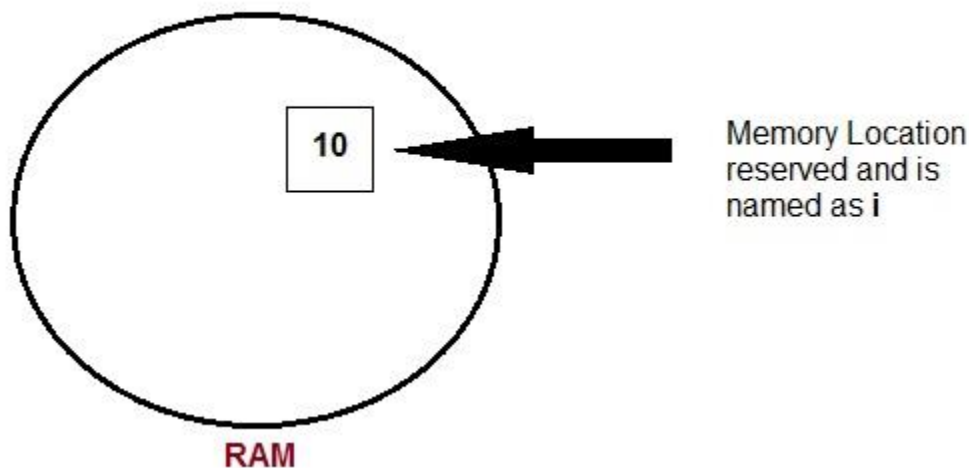
Basic types of Variables

Each variable while declaration must be given a data type, on which the memory assigned to the variable, depends. Following are the basic types of variables,

bool - For variable to store boolean values(True or False)

char - For variables to store character types.

int - for variable with integral values

float and double are also types for variables with large and floating point values

Declaration and Initialization:

Variable must be declared before they are used. Usually it is preferred to declare them at the starting of the program, but in C++ they can be declared in the middle of program too, but must be done before using them.

*Example***:**

int i;   // declared but not initialized

char c;

int i, j, k; // Multiple declaration

Initialization means assigning value to an already declared variable,

int i;   // declaration

i = 10; // initialization

Initialization and declaration can be done in one single step also,

int i=10;   //initialization and declaration in same step

int i=10, j=11;

If a variable is declared and not initialized by default it will hold a garbage value. Also, if a variable is once declared and if try to declare it again, we will get a compile time error.

int i,j;

i=10;

j=20;

int j=i+j; //compile time error, cannot redeclare a variable in same scope

Scope of Variables

All the variables have their area of functioning, and out of that boundary they don't hold their value, this boundary is called scope of the variable. For most of the cases its between the curly braces,in which variable is declared that a variable exists, not outside it. We will study the storage classes later, but as of now, we can broadly divide variables into two main types,

- Global Variables
- Local variables

Global variables:

Global variables are those, which are once declared and can be used throughout the lifetime of the program by any class or any function. They must be declared outside the main() function. If only declared, they can be assigned different values at different time in program lifetime. But even if they are declared and initialized at the same time outside the main() function, then also they can be assigned any value at any point in the program.

*Example***:** Only declared, not initialized

include<iostream.h>

```
int x;            // Global variable declared
int main()
{
x=10;             // Initialized once
cout <<"first value of x = "<< x;
x=20;             // Initialized again
cout <<"Initialized again with value = "<< x;
}
```

Local Variables:

Local variables are the variables which exist only between the curly braces, in which its declared. Outside that they are unavailable and lead to compile time error.

Example:

include <iostream.h>

int main()

{

int i=10;

if(i<20)   // if condition scope starts

{

int n=100;   // Local variable declared and initialized

}   // if condition scope ends

cout << n;   // Compile time error, n not available here

}

Constants:

Constants refer to fixed values that the program may not alter and they are called **literals**. Constants can be of any of the basic data types and can be divided into Integer Numerals, Floating-Point Numerals, Characters, Strings and Boolean Values. Again, constants are treated just like regular variables except that their values cannot be modified after their definition.

Syntax:

const type constant_name;

Integer literals:

An integer literal can be a decimal, octal, or hexadecimal constant. A prefix specifies the base or radix: 0x or 0X for hexadecimal, 0 for octal, and nothing for decimal.

An integer literal can also have a suffix that is a combination of U and L, for unsigned and long, respectively. The suffix can be uppercase or lowercase and can be in any order.

Here are some examples of integer literals:

212   // Legal

215u   // Legal

0xFeeL   // Legal

078   // Illegal: 8 is not an octal digit

032UU   // Illegal: cannot repeat a suffix

Floating-point literals:

A floating-point literal has an integer part, a decimal point, a fractional part, and an exponent part. You can represent floating point literals either in decimal form or exponential form.

While representing using decimal form, you must include the decimal point, the exponent, or both and while representing using exponential form, you must include the integer part, the fractional part, or both. The signed exponent is introduced by e or E.

Here are some examples of floating-point literals:

3.14159   // Legal

314159E-5L   // Legal

510E   // Illegal: incomplete exponent

210f   // Illegal: no decimal or exponent

.e55   // Illegal: missing integer or fraction

Boolean literals:

There are two Boolean literals and they are part of standard C++ keywords:

- A value of **true** representing true.
- A value of **false** representing false.

You should not consider the value of true equal to 1 and value of false equal to 0.

Character literals:

Character literals are enclosed in single quotes. If the literal begins with L (uppercase only), it is a wide character literal (e.g., L'x') and should be stored in **wchar_t** type of variable. Otherwise, it is a narrow character literal (e.g., 'x') and can be stored in a simple variable of **char** type.

A character literal can be a plain character (e.g., 'x'), an escape sequence (e.g., '\t'), or a universal character (e.g., '\u02C0').

There are certain characters in C++ when they are preceded by a backslash they will have special meaning and they are used to represent like newline (\n) or tab (\t). Here, you have a list of some of such escape sequence codes:

Following is the example to show few escape sequence characters:

Escape sequence	Meaning
\\	\ character
\'	' character
\"	" character
\?	? character
\a	Alert or bell
\b	Backspace
\f	Form feed
\n	Newline
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\ooo	Octal number of one to three digits
\xhh . . .	Hexadecimal number of one or more digits

```
#include <iostream.h>
int main()
{
cout <<"Hello\tWorld\n\n";
return 0;
}
```

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

Hello   World

String literals:

String literals are enclosed in double quotes. A string contains characters that are similar to character literals: plain characters, escape sequences, and universal characters.

You can break a long line into multiple lines using string literals and separate them using whitespaces.

Here are some examples of string literals. All the three forms are identical strings.

"hello, dear"

"hello, \

dear"

"hello, ""d""ear"

The const Keyword:

You can use **const** prefix to declare constants with a specific type as follows:

Syntax:

const type variable = value;   [or]   type const variable=value;

Following example explains it in detail:

```
#include <iostream.h>
int main()
{
const int  LENGTH = 10;
const int  WIDTH  = 5;
const char NEWLINE = '\n';
int area;
```

area = LENGTH * WIDTH;

cout << area;

cout << NEWLINE;

return 0;

}

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

50

Note that it is a good programming practice to define constants in CAPITALS.

Operators:

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. C++ is rich in built-in operators and provides the following types of operators:

- Arithmetic Operators
- Logical Operators
- Bitwise Operators

Arithmetic Operators:

There are following arithmetic operators supported by C++ language:

Operator	Description
+	Adds two operands
-	Subtracts second operand from the first
*	Multiplies both operands
/	Divides numerator by de-numerator
%	Modulus Operator and remainder of after an integer division
++	Increment operator, increases integer value by one
--	Decrement operator, decreases integer value by one

Arithmetic operator example:

include<iostream.h>

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;   }

Output:

Enter 2 integers: 8   4

The sum of 4 and 8 is 12.

The average of 4 and 8 is 6.

Logical Operators:

There are following logical operators supported by C++ language

Operator   Description

&&   Called Logical AND operator. If both the operands are non-zero, then condition becomes

true.   (A && B) is false.

|| Called Logical OR Operator. If any of the two operands is non-zero, then condition

       becomes true. (A || B) is true.

!   Called Logical NOT Operator. Use to reverses the logical state of its operand. If a

       condition is true, then Logical NOT operator will make false.   !(A && B) is true.

Logical operator example:

#include<iostream.h>

void main()

{   int a, b;

cout<<"\n Enter the a and b values:";

```
cin>>a>>b;
if(a<b)&&(b>a)
cout<<"A is small";
else
cout<<"B is big";  }
```

Output:

1) Enter the a and b values: 100   300   2) Enter the a and b values: 1   3

    A is small   B is big

Bitwise Operators:

Bitwise operator works on bits and performs bit-by-bit operation. The truth tables for &, |, and ^ are as follows:

p	q	p & q	p | q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

Operator	Description
<<	Binary Left Shift Operator
>>	Binary Right Shift Operator
~	Binary Ones Complement Operator
&	Binary AND Operator
^	Binary XOR Operator
|	Binary OR Operator

Bitwise operator example: **Output:**

```
#include <iostream.h>
void main() {
unsigned int a = 60;          // 60 = 0011 1100
unsigned int b = 13;          // 13 = 0000 1101
int c = 0;
c = a & b;          // 12 = 0000 1100
cout << "Line 1 - Value of c is : " << c << endl ;
c = a | b;          // 61 = 0011 1101
cout << "Line 2 - Value of c is: " << c << endl ;
c = a ^ b;          // 49 = 0011 0001
cout << "Line 3 - Value of c is: " << c << endl ;
c = ~a;          // -61 = 1100 0011
cout << "Line 4 - Value of c is: " << c << endl;  }
```

Line 1 - Value of c is: 12
Line 2 - Value of c is: 61
Line 3 - Value of c is: 49
Line 4 - Value of c is: -61

C++ Basic Input/Output:

C++ I/O occurs in streams, which are sequences of bytes. If bytes flows from a device like a keyboard, a disk drive, or a network connection etc. to main memory, this is called **input operation** and if bytes flow from main memory to a device likes a display screen, a printer, a disk drive, or a network connection, etc, this is called **output operation**.

I/O Library Header Files:

There are following header files important to C++ programs:

Header File	Function and Description
<iostream>	This file defines the cin, cout, objects, which correspond to the standard input stream, the standard output stream.
<stdio>	This files defines the printf(), scanf() functions, which correspond to the standard input, the standard output
<conio>	This header declares several useful library functions for performing "console input and output" from a program like clrscr(),getch() functions.

The standard output stream (cout):

The predefined object **cout** is an instance of **ostream** class. The cout object is said to be "connected to" the standard output device, which usually is the display screen. The **cout** is used in conjunction with the stream insertion operator, which is written as << which are two less than signs as shown in the following example.

```
#include <iostream.h>
int main( )
{
char str[] = "Hello C++";

cout <<"Value of str is : "<< str << endl;
}
```

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

Value of str is : Hello C++

The C++ compiler also determines the data type of variable to be output and selects the appropriate stream insertion operator to display the value. The << operator is overloaded to output data items of built-in types integer, float, double, strings and pointer values.

The insertion operator << may be used more than once in a single statement as shown above and **endl** is used to add a new-line at the end of the line.

The standard input stream (cin):

The predefined object **cin** is an instance of **istream** class. The cin object is said to be attached to the standard input device, which usually is the keyboard. The **cin** is used in conjunction with the stream extraction operator, which is written as >> which are two greater than signs as shown in the following example.

```
#include <iostream>
int main( )
{
char name[50];
```

cout <<"Please enter your name: ";

cin >> name;

cout <<"Your name is: "<< name << endl;

}

When the above code is compiled and executed, it will prompt you to enter a name. You enter a value and then hit enter to see the result something as follows:

Please enter your name: cplusplus

Your name is: cplusplus

The C++ compiler also determines the data type of the entered value and selects the appropriate stream extraction operator to extract the value and store it in the given variables.

The stream extraction operator >> may be used more than once in a single statement. To request more than one datum you can use the following:

cin >> name >> age;

This will be equivalent to the following two statements:

cin >> name;

cin >> age;

Printf and Scanf:

Printf():

Printf is a predefined function in "stdio.h" header file, by using this function, we can print the data or user defined message on console or monitor. While working with printf(), it can take any number of arguments but first argument must be within the double cotes (" ") and every argument should separated with comma (,) Within the double cotes, whatever we pass, it prints same, if any format specifies are there, then that copy the type of value. The scientific name of the monitor is called console.

Syntax:

    printf("user defined message");

Syntax:

prinf("Format specifers",value1,value2,..);

Example of printf() function:

int a=10;

double d=13.4;

printf("%f%d",d,a);

scanf():

scanf() is a predefined function in "stdio.h" header file. It can be used to read the input value from the keyword.

Syntax:

scanf("format specifiers",&value1,&value2,.....);

Example of scanf function:

int a;

float b;

scanf("%d%f",&a,&b);

In the above syntax format specifier is a special character in the C language used to specify the data type of value.

Format specifier:

Format specifier	Type of value
%d	Integer
%f	Float
%lf	Double
%c	Single character
%s	String
%u	Unsigned int
%ld	Long int

%lf	Long double

The address list represents the address of variables in which the value will be stored.

Example:

int a;

float b;

scanf("%d%f",&a,&b);

In the above example scanf() is able to read two input values (both int and float value) and those are stored in a and b variable respectively.

Syntax :

double d=17.8;

char c;

long int l;

scanf("%c%lf%ld",&c&d&l);

Comments:

Comment in C++ Programming is similar as that of in C .Each and every language will provide this great feature which is used to document source code. We can create more readable and eye catching program structure using comments. We should use as many as comments in C++ program. **Comment is non executable Statement in the C++.**

Types of Comment in C++ Programming:

We can have two types of comment in Programming –

1. Single Line Comment   - //
2. Multiple Line Comment - /*   */

Single Line Comments in C++

cout<<"Hello"; //Print Hello Word

cout<<"www.c4learn.com"; //Website

cout<<"Pritesh Taral"; //Author

Multiple Line Comments in C++

int main()

{

/* this comment

  can be considered

   as

 multiple line comment */

cout << "Hello C++ Programming";

return(0);}

Character - Input & Output:

The getchar() and putchar() Functions:

The **int getchar(void)** function reads the next available character from the screen and returns it as an integer. This function reads only single character at a time. You can use this method in the loop in case you want to read more than one character from the screen.

The **int putchar(int c)** function puts the passed character on the screen and returns the same character. This function puts only single character at a time. You can use this method in the loop in case you want to display more than one character on the screen. Check the following example

```
#include <stdio.h>
int main( ) {

int c;

printf( "Enter a value :");
c = getchar( );

printf( "\nYou entered: ");
putchar( c );

return 0;
```

}

When the above code is compiled and executed, it waits for you to input some text. When you enter a text and press enter, then the program proceeds and reads only a single character and displays it as follows −

Enter a value: this is test

You entered: t

getc(), putc():

getc(), putc() functions are file handling function in C programming language which is used to read a character from a file (getc) and display on standard output or write into a file (putc). Please find below the description and syntax for above file handling functions.

File operation	Declaration & Description
getc()	Declaration: int getc(FILE *fp) getc functions is used to read a character from a file. In a C program, we read a character asbelow. **getc** (fp);
putc()	Declaration: int putc(int char, FILE *fp) putc function is used to display a character on standard output or is used to write into a file. In a C program, we can use putc as below. putc(char,stdout); putc(char, fp);

Control structures form the basic entities of a "**structured programming language**". We all know languages like C/C++ or Java are all structured programming languages. **Control structures are used to alter the flow of execution of the program.** Why do we need to alter the program flow? The reason is "**decision making**"! In life, we may be given with a set of option like doing "Electronics" or "Computer science". We do make a decision by analyzing certain conditions (like our personal interest, scope of job opportunities etc). With the decision we make, we alter the flow of our life's direction. This is exactly what happens in a C/C++ program. We use control structures to make decisions and alter the direction of program flow in one or the other path(s) available.

There are **three types** of control structures available in C and C++

1) Sequence structure (straight line paths)

2) Selection structure (one or many branches)

3) Loop structure (repetition of a set of activities)

All the 3 control structures and its flow of execution are represented in the flow charts given below.

Sequence Structure
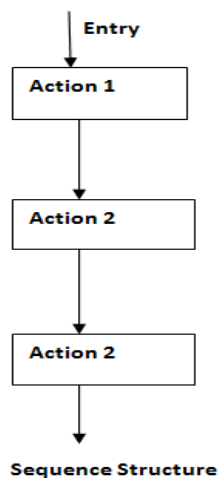
Control statements in C/C++ to implement control structures

We have to keep in mind one important fact:- all program processes can be implemented with these 3 control structures only. That's why I wrote "**control structures are the basic entities of a structured programming language**". To implements these "control structures" in a C/C++

program, the language provides 'control statements'. So to implement a particular control structure in a programming language, we need to learn how to use the relevant control statements in that particular language.

The control statements are:-

- **Switch**
- **If**
- **If Else**
- **While**
- **Do While**
- **For**

As shown in the flow charts:-

- Selection structures are implemented using **If**, **If Else** and **Switch** statements.
- Looping structures are implemented using **While**, **Do While** and **For** statements.

Selection structures
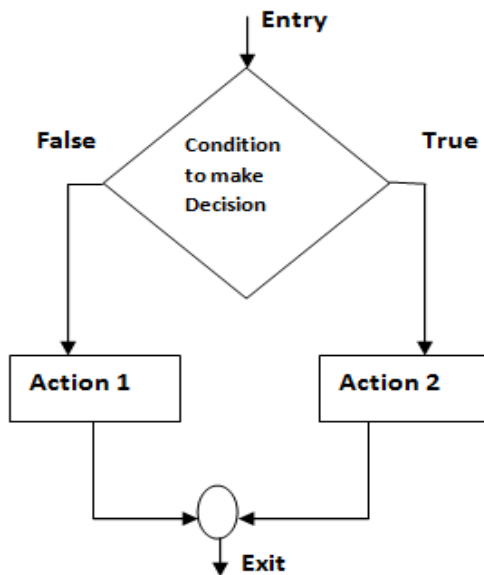
Selection Structure

Implemented using:- If and If..else control statements

switch is used for multi branching

Selection structure:

Selection structures are used to perform 'decision **making**'and then branch the program flow based on the outcome of decision making. Selection structures are implemented in C/C++ with If, If Else and Switch statements. If and If Else statements are 2 way branching statements where as Switch is a multi branching statement.

Simple if statement:

The syntax format of a simple if statement is as shown below:

if (expression) // This expression is evaluated. If expression is TRUE statements inside the braces will be executed

{

statement 1;

statement 2;

}

statement 1;// Program control is transferred directly to this line, if the expression is FALSE

statement 2;

The expression given inside the brackets after **if** is evaluated first. If the expression is true, then statements inside the curly braces that follow if(expression) will be executed. If the expression is false, the statements inside curly braces will not be executed and program control goes directly to statements after curly braces.

Example:	**Output:**
int main()	m and n are equal
{	
int m=40,n=40;	
if (m == n)	
{	
cout<<"m and n are equal";	
}	
}	

The If Else statement:

Syntax format for If Else statement is shown below:

if(expression 1)// Expression 1 is evaluated. If TRUE, statements inside the curly braces are

executed.

{ //If FALSE program control is transferred to immedate else if statement.

statement 1;

statement 2;

}

else if(expression 2)// If expression 1 is FALSE, expression 2 is evaluated.

{

statement 1;

statement 2;

}

else if(expression 3) // If expression 2 is FALSE, expression 3 is evaluated

{

statement 1;

statement 2;

}

else // If all expressions (1, 2 and 3) are FALSE, the statements that follow this else (inside curly

braces) is executed.

{

statement 1;

statement 2;

}

other statements;

The execution begins by evaluation expression 1. If it is **TRUE**, then statements inside the immediate curly braces is evaluated. If it is **FALSE**, program control is transferred directly to immediate else if statement. Here expression 2 is evaluated for TRUE or FALSE. The process continues. If all expressions inside the different if and else if statements are FALSE, then the last

else statement (without any expression) is executed along with the statements 1 and 2 inside the curly braces of last **else** statement.

Example program to demo "If Else":

#include <iostream.h>

int main()

{

 int m=40,n=20;

 if (m == n)

 {

 cout<<"m and n are equal";

 }

 else

 {

 cout<<"m and n are not equal";

 }return o;

 }

Switch statement:

Switch is a multi branching control statement.

Syntax for switch statement is shown below:

switch(expression)   // Expression is evaluated. The outcome of the expression should be an integer or a character constant

{

case value1: // case is the keyword used to match the integer/character constant from expression.

            //value1, value2 ... are different possible values that can come in expression

statement 1;

statement 2;

break;   // break is a keyword used to break the program control from switch block.

case value2:

statement 1;

statement 2;

break;

default: // default is a keyword used to execute a set of statements inside switch, if no case values match the expression value.

statement 1;

statement 2;

break;

}

Execution of switch statement begins by evaluating the expression inside the switch keyword brackets. The expression should be an integer (1, 2, 100, 57 etc) or a character constant like **'a'**, **'b'** etc. This expression's value is then matched with each case values. There can be any number of case values inside a switch statements block. If first case value is not matched with the expression value, program control moves to next case value and so on. **When a case value matches with expression value, the statements that belong to a particular case value are executed.**

Notice that last set of lines that begins with **default**. The word **default** is a **keyword in C/C++**. When used inside switch block, it is intended to execute a set of statements, if no case values matches with expression value. So if no case values are matched with expression value, the set of statements that follow **default:** will get executed.

Note: Notice the **break** statement used at the end of each case values set of statements. The word break is a **keyword in C++** used to break from a block of curly braces. The switch block has two curly braces { }. The keyword break causes program control to exit from switch block.

Example program to demo working of "switch":	**Output:1**
#include<iostream.h>	Hello user, Enter a number
void main()	3
{	INDIA
int num;	
cout<<"Hello user, Enter a number";	**Output:2**
cin>>num; // Collects the number from user	Hello user, Enter a number
switch(num)	4
{	WRONG ENTRY
case 1:	

cout<<"UNITED STATES";

break;

case 2:

cout<<"SPAIN";

break;

case 3:

cout<<"INDIA";

default:

cout<<"WRONG ENTRY";

}

}

Note:- Switch statement is used for multiple branching. The same can be implemented using **nested "If Else"** statements. But use of nested if else statements make program writing tedious and complex. Switch makes it much easier. Compare this program with above one.
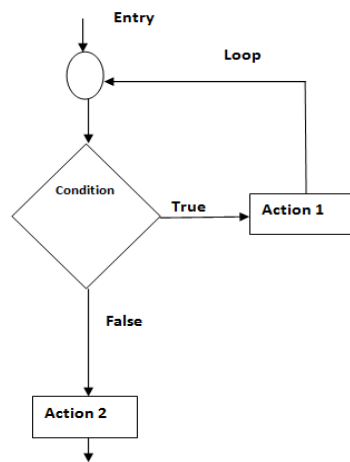
Loop structures:

A loop structure is used to execute a certain set of actions for a predefined number of times or until a particular condition is satisfied. There are 3 control statements available in C/C++ to implement loop structures. **While, Do while and For statements.**

The while statement:

Syntax for while loop is shown below:

while(condition)// This condition is tested for TRUE or FALSE. Statements inside curly braces are executed as long as condition is TRUE

{

statement 1;

statement 2;

statement 3;

}

The condition is checked for TRUE first. If it is TRUE then all statements inside curly braces are executed.Then program control comes back to check the condition has changed or to check if it is still TRUE. The statements inside braces are executed repeatedly, as long as the condition is TRUE. When the condition turns FALSE, program control exits from while loop.

Note:- while is an entry controlled loop. Statement inside braces are allowed to execute only if condition inside while is TRUE.

Example program to demo working of "while loop"

An example program to collect a number from user and then print all numbers from zero to that particular collected number is shown below. That is, if user enters 10 as input, then numbers from 0 to 10 will be printed on screen.

Note:- The same problem is used to develop programs for do while and for loops

```
#include<iostream.h>

void main()

{

int num;

int count=0;            // count is initialized as zero to start printing from zero.

cout<<"Hello user, Enter a number";

cin>>num;              // Collects the number from user

while(count<=num)   // Checks the condition - if value of count has reached value of num or not.

{ cout<<count;

count=count+1;        // value of count is incremented by 1 to print next number.
```

} }

The do while statement:

Syntax for do while loop is shown below:

do

{

statement 1;

statement 2;

statement 3;

}

while(condition);

Unlike while, do while is an exit controlled loop. Here the set of statements inside braces are executed first. The condition inside while is checked only after finishing the first time execution of statements inside braces. If the condition is TRUE, then statements are executed again. This process continues as long as condition is TRUE. Program control exits the loop once the condition turns FALSE.

Example program to demo working of "do while":

```
#include<iostream.h>
void main()
{
int num;
int count=0;        // count is initialized as zero to start printing from zero.
cout<<"Hello user, Enter a number";
cin>>num;          // Collects the number from user
do
{
cout<<count;    // Here value of count is printed for one time intially and then only condition is checked.
count=count+1; // value of count is incremented by 1 to print next number.
}while(count<=num); }
```

The for statement:

Syntax of for statement is shown below:

for(initialization statements;test condition;iteration statements)

{

statement 1;

statement 2;

statement 3;

}

The for statement is an entry controlled loop. The difference between while and for is in the number of repetitions. The for loop is used when an action is to be executed for a predefined number of times. The while loop is used when the number of repetitions is not predefined.

Working of for loop:

The program control enters the for loop. At first it executes the statements given as initialization statements. Then the condition statement is evaluated. If conditions are TRUE, then the block of statements inside curly braces is executed. After executing curly brace statements fully, the control moves to the "iteration" statements. After executing iteration statements, control comes back to condition statements. Condition statements are evaluated again for TRUE or FALSE. If TRUE the curly brace statements are executed. This process continues until the condition turns FALSE.

Note 1:- The statements given as "**initialization statements**" are executed only once, at the beginning of a for loop.

Note 2: There are 3 statements given to a for loop as shown. One for initialization purpose, other for condition testing and last one for iterating the loop. Each of these 3 statements are separated by semicolons.

Example program to demo working of "for loop":

```
#include<iostream.h>
void main()
{
int num,count;
cout<<"Hello user, Enter a number";
cin>>num;      // Collects the number from user
for(count=0;count<=num;count++)   // count is initialized to zero inside for statement. The condition is
checked and statements are executed.
{ cout<<count);   // Values from 0 are printed.
```

} }

C++ Jump Statements:

The jump statements unconditionally transfer program control within a function. C++ has four statements that perform an unconditional branch:

- break
- continue

Of these, you may use return and goto anywhere in the program whereas break and continue are used inside smallest enclosing like loops etc. In addition to the above four, C++ provides a standard library function exit() that helps you break out of a program. The return statement is used to return from a function.

C++ break Statement

The break statement enables a program to skip over part of the code. A break statement terminates the smallest enclosing while, do-while, for, or switch statement. Execution resumes at the statement immediately following the body of the terminated statement.

The following code fragment gives you an example of a break statement:

```
:
:
int a, b, c, i;
for(i=0; i<20; i++)
{
cout <<"Enter 2 numbers" ;
cin >> a >> b ;
if(b == 0)
break;
else
c = a/b ;
cout <<"\n Quotient ="<< c <<"\n" ;
```

The above code fragment inputs two numbers. If the number b is zero, the loop immediately terminated otherwise the numbers are repeated input and their quotients are displayed.

If a break statement appears in a nested-loop structure, then it causes an exit from only the very loop it appears in.

For example:

:

:

for(i=0; i<10; i++)

{

j=0;

cout <<"\n Enter character";

cin >> ch;

cout <<"\n";

for(; ;)

{

cout << ch;

j++ ;

if(j == 10)

break;

}

cout <<"\n...." ;

}

The above code fragment inputs a character and prints it 10 times. The inner loop has an infinite loop structure but the break statement terminates it as soon as j becomes 10 and the control comes to the statement following the inner loop which prints a line of dashes.

A break used in switch statement will affect only that switch i.e., It will terminate only the very switch it appears in. It does not affect any loop the switch happens to be in.

C++ continue Statement:

The continue is another jump statement like the break statement as both the statements skip over a part of the code. But the continue statement is somewhat different from break. Instead of forcing termination, it forces the next iteration of the loop to take place, skipping any code between.

For the for loop, continue causes the next iteration by updating the variable and then causing the test-expression's evaluation. For the while and do-while loops, the program control passes to the conditional tests.

Note - The continue statement skips the rest of the loop statements and causes the next iteration of the loop. The following code fragment gives you an example of continue statement :

:

```
int a, b, c, i;
for(i=0; i<20; i++)
{
cout <<"\n Enter 2 numbers" ;
cin >> a >> b ;
if(b == 0)
{
cout <<"\n The denominator cannot be zero"<<"Enter again !";
continue;
}
else
c = a/b ;
cout <<"\n Quotient ="<< c <<"\n" ;
}
```

Sometimes you need to abandon iteration of a loop prematurely. Both the statements break and continue can help in that but in different situations.

Tip - Do not confuse the break (exits the block) and continue (exits the remaining statement(s)) statements. A break statement inside a loop will abort the loop and transfer control to the statement following the loop. A continue statement will just abandon the current iteration and let the loop start the next iteration.
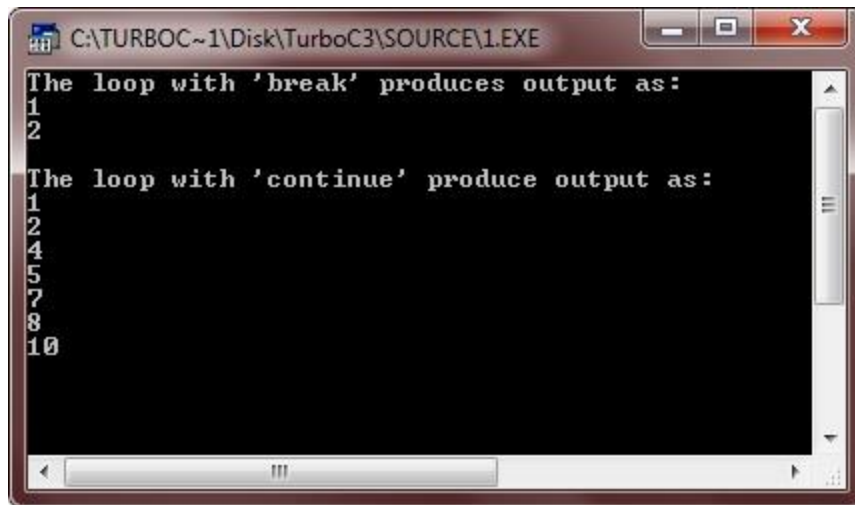
C++ break and continue Statement Example:

Following example program uses two loops to perform the same thing, but replaces break statement with continue. Have a look at one code and then the output to understand the difference between break and continue statements:

/* C++ Jump Statements - C++ break and continue Statement */

```
#include<iostream.h>

#include<conio.h>

void main()

{

clrscr();

cout<<"The loop with \'break\' produces output as:\n";

for(int i=1; i<=10; i++)

{

if((i%3)==0)

break;

else

cout<<i<<endl;

}

cout<<"\nThe loop with \'continue\' produce output as:\n";

for(i=1; i<=10; i++)

{

if((i%3)==0)

continue;

else

cout<<i<<endl;

}

getch();

}
```

When the C++ program is compiling and executed, it will produce the following output:

POSSIBLE QUESTIONS

UNIT I

Part-A

Online Examinations   (One marks)

1. _____ is the non-structure language or object oriented language

    a) Objects   b) classes   c) c++   **d) c language**

2. The _____ is an exit-controlled loop

    a) while   **b) do-while**   c) for   d) switch

3. Where does the execution of the program starts?

    a) user-defined function   **b) main function**   c) void function   d) none

4. What are mandatory parts in function declaration?

    a) return type, function name   b) return type, function name, parameters

    c) Both a and b   d) none of the mentioned

5. Which is more effective while calling the functions?

    a) Call by value   **b) call by reference** c) call by pointer   d) none

6. Which of the following correctly declares an array?

    a) int array[10];   b) int array;   c) array{10};   d) array array[10];

7. What is the index number of the last element of an array with 9 elements?

    a) 9   **b) 8**   c) 0   d) Programmer-defined

8. A structure can have both variable and functions as _____

    a) Objects   b) classes   **c) members**   d) arguments

9. Pointer is

    a) A keyword used to create variables   b) A variable that stores address of an instruction

    c) A variable that stores address of other variable   d) All of the above

10. Which value we cannot assign to reference?

    a) Integer   b) floating   c) unsigned   **d) null**

PART B- 2 MARKS

1. Define variables?
2. How many times the following loop is executed?

   int s=0,i=0;

   while(i++<5)

   s+=i;

3. What is Keyword?
4. Differentiate between do-while and while.
5. Define data type.
6. Write about break and continue in loops.
7. Give the difference between procedure oriented and object oriented programming.
8. Give the steps to compile and execute a C program.
9. With syntax and example explain all the different forms of if statement.
10. List the primary data types and give examples for each.
11. What is for loop? Give syntax. Explain it with example.
12. Explain in detail about various types of operators. Provide examples for each.

PART C- 6 MARKS

1. Describe in detail about conditional statements in c++ with example program.

2. Explain with an example   (i) Operators (ii) formatted and console I/O.

3. Explain looping statements with example program?

4. Describe about Constants and Keywords with example.

5. Explain While and do-While statement with example program?

6. What are the different ways of writing comment lines in C++?

7. Discuss the role of formatted console I/O operations with example.

8. What is operator? Explain any two operators with example.

9. Write note on   i) stdio.h   ii) iostream.h   iii) conio.h

10. Give the use of comments in C program.

11. What is while loop? Give syntax. Explain it with an example program.

12. With syntax and example explain the character Input/Output methods with example.

13. What are the functions used in Formatted I/O? Explain in detail with examples.

14. Evaluate the expression using operator precedence, C=((10+5)*(6/3)/(10-5)).

15. What is the use of switch statement? Give its syntax and explain with an example.

16. Write a program to calculate factorial of a given number.

17. Write a program print the fibonacci series.

Part -A Online Examinations   (1 mark questions)

SUBJECT: PROGRAMMING FUNDAMENTALS USING C/C++   **SUBJECT CODE: 17CSU101**

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 or explanations	encapsulation	inheritance	Dynamic binding	Abstraction	Abstraction

#	Question					
8	Attributes are sometimes called_____	data members	methods	messages	functions	data members
9	The functions operate on the datas 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	none	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	none	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 t pass on t 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	:	;	,	++	;

#	Question					
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	none	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)	none	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	none	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	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
54	When the data is passed to the output devices it is called_____	Source stream	Object stream	Destination stream	Input stream.	Destination stream
55	The C++ have a number of stream classes that are used to work with _____ operations.	Console I/O	Console and file	formatted console	unformatted console	Console and file

56	The data accepted with default setting by I/O function of the language is known as-----	Formatted	Unformatted	Argumented	files	Unformatted
57	_____ is used as the input stream to read data.	Cout	Printf	Cin	Scanf.	Cin
58	cin and cout are _____ for input and output of data.	user defined stream	system defined stream	Pre defined stream	stream	Pre defined stream
59	.The data obtained or represented with some manipulators are called _____.	formatted data	unformatted data	extracted data	None.	formatted data
60	The output formats can be controlled with manipulators having the header file as	iostream.h	conio.h	stdlib.h	iomanip.h	iomanip.h
61	The _____ and _____ are derived classes from ios based class.	istream and ostream	source and destination stream	iostream and source stream	None.	istream and ostream
62	The manipulator << endl is equivalent to____	'\t'	'\r'	'\n'	'\b'	'\n'
63	While loop is _____ statement.	entry controlled	exit controlled	branching	none	entry controlled
64	Do.. While loop is _____ statement.	entry controlled	exit controlled	branching	none	exit controlled
65	For loop is an _____ statement.	entry controlled	exit controlled	branching	none	entry controlled
66	_____ statement causes loop to be continued with next	continue	goto	break	exit	continue

KARPAGAM ACADEMY OF HIGHER EDUCATION

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

Coimbatore - 641021.

(For the candidates admitted from 2017 onwards)

DEPARTMENT OF COMPUTER SCIENCE,CA & IT

SUBJECT   : **PROGRAMMING FUNDAMENTALS USING C/C++**

SEMESTER   : **I**   **L T P C**

SUBJECT CODE: **17CSU101**   **CLASS**   : **I B.Sc.CS**   **4 0 0 4**

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.

Suggested Readings:

1. Herbtz Schildt, 2003,*C++: The Complete Reference*, 4th Edition, McGraw Hill.

2. Bjarne Stroustrup, 2013, *The C++ Programming Language,* 4th Edition, Addison-Wesley.

3. Bjarne Stroustroup, 2014, *Programming - Principles and Practice using C++,* 2nd Edition, Addison-Wesley.

4. E Balaguruswamy, 2008,*Object Oriented Programming with C++,* 2nd Edition ,Tata McGraw-Hill Education.

5. Paul Deitel, Harvey Deitel, 2011,*C++ How to Program*, 8th Edition, Prentice Hall.

6. John R. Hubbard, 2000, *Programming with C++,* 2nd Edition ,Schaum's Series.

7. Stefan Bjornander, 2016,*C++ Windows Programming,* Published by Packt Publishing Ltd.

8. Scott Meyers, 2005, *Effective C++,* 3rd Edition, Published by Addison-Wesley.

9. Harry, H. Chaudhary, 2014 ,*Head First C++ Programming: The Definitive Beginner's Guide*, First Create space Inc, 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*, 5th Edition, Published by Addison-Wesley.

12. Debasish Jana , 2014,*C++ And Object-Oriented Programming Paradigm*,Published by PHI Learning Pvt. Ltd.

13. Richard L. Stegman, 2016, *Focus on Object-oriented Programming With C++,*6th Edition ,CreateSpace Independent Publishing Platform,.

14. Andrew Koeni, Barbara, E. Moo,2000,*Accelerated using C++,* Published by Addison-Wesley .

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. http://*www.cplusplus.com/*

6. http://*www.cppreference.com/*

UNIT 2

Functions

Functions allow to structure programs in segments of code to perform individual tasks.

In C++, a function is a group of statements that is given a name, and which can be called from some point of the program. The most common syntax to define a function is:

type name (parameter1, parameter2, ...) { statements }

Where:

- type is the type of the value returned by the function.
- name is the identifier by which the function can be called.
- parameters (as many as needed): Each parameter consists of a type followed by an identifier, with each parameter being separated from the next by a comma. Each parameter looks very much like a regular variable declaration (for example:int x), and in fact acts within the function as a regular variable which is local to the function. The purpose of parameters is to allow passing arguments to the function from the location where it is called from.
- statements is the function's body. It is a block of statements surrounded by braces { } that specify what the function actually does.

Call by value

The call by value method of passing arguments to a function copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.

By default, C++ uses call by value to pass arguments. In general, this means that code within a function cannot alter the arguments used to call the function. Consider the function **swap()** definition as follows.

```
// function definition to swap the values.
void swap(int x, int y)
{
int temp;
  temp = x; /* save the value of x */
  x = y;   /* put y into x */
```

y = temp; /* put x into y */

  return;

}

 Now, let us call the function **swap()** by passing actual values as in the following example:

```cpp
#include <iostream>
using namespace std;
// function declaration
void swap(int x, int y);
int main ()
{
  // local variable declaration:
int a = 100;
int b = 200;
cout<< "Before swap, value of a :" << a <<endl;
cout<< "Before swap, value of b :" << b <<endl;
  // calling a function to swap the values.
swap(a, b);
cout<< "After swap, value of a :" << a <<endl;
cout<< "After swap, value of b :" << b <<endl;
  return 0;
}
```

When the above code is put together in a file, compiled and executed, it produces the following result:

Before swap, value of a :100

Before swap, value of b :200

After swap, value of a :100

After swap, value of b :200

Which shows that there is no change in the values though they had been changed inside the function.

Call by reference

The **call by reference** method of passing arguments to a function 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 passed argument.

To pass the value by reference, argument reference is passed to the functions just like any other value. So accordingly you need to declare the function parameters as reference types as in the following function **swap()**, which exchanges the values of the two integer variables pointed to by its arguments.

```
// function definition to swap the values.
void swap(int&x, int&y)
{
int temp;
   temp = x; /* save the value at address x */
   x = y;    /* put y into x */
   y = temp; /* put x into y */

   return;
}
```

For now, let us call the function **swap()** by passing values by reference as in the following example:

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

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

int main ()
{
   // local variable declaration:
```

```
int a = 100;
int b = 200;
cout<< "Before swap, value of a :" << a <<endl;
cout<< "Before swap, value of b :" << b <<endl;
   /* calling a function to swap the values using variable reference.*/
swap(a, b);
cout<< "After swap, value of a :" << a <<endl;
cout<< "After swap, value of b :" << b <<endl;
   return 0;
}
```

When the above code is put together in a file, compiled and executed, it produces the following result:

```
Before swap, value of a :100
Before swap, value of b :200
After swap, value of a :200
After swap, value of b :100
```

C++ Inline Functions

C++ **inline** function is powerful concept that is commonly used with classes. If a function is inline, the compiler places a copy of the code of that function at each point where the function is called at compile time.

Any change to an inline function could require all clients of the function to be recompiled because compiler would need to replace all the code once again otherwise it will continue with old functionality.

To inline a function, place the keyword **inline** before the function name and define the function before any calls are made to the function. The compiler can ignore the inline qualifier in case defined function is more than a line.

A function definition in a class definition is an inline function definition, even without the use of the **inline** specifier.

Following is an example, which makes use of inline function to return max of two numbers:

```
#include <iostream>
```

```
using namespace std;
inline intMax(int x, int y)
{
   return (x > y)? x : y;
}
// Main function for the program
intmain( )
{
cout<< "Max (20,10): " <<Max(20,10) <<endl;
cout<< "Max (0,200): " <<Max(0,200) <<endl;
cout<< "Max (100,1010): " <<Max(100,1010) <<endl;
   return 0;
}
```

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

Max (20,10): 20

Max (0,200): 200

Max (100,1010): 1010

Let's have a look at an example:

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

int addition (int a, int b)
{
int r;
 r=a+b;
 return r;
}
```

int main ()

{

int z;

  z = addition (5,3);

cout<< "The result is " << z;

}

The result is 8

This program is divided in two functions: addition and main. Remember that no matter the order in which they are defined, a C++ program always starts by calling main. In fact, main is the only function called automatically, and the code in any other function is only executed if its function is called from main (directly or indirectly).

In the example above, main begins by declaring the variable z of type int, and right after that, it performs the first function call: it calls addition. The call to a function follows a structure very similar to its declaration. In the example above, the call to addition can be compared to its definition just a few lines earlier:

The parameters in the function declaration have a clear correspondence to the arguments passed in the function call. The call passes two values, 5 and 3, to the function; these correspond to the parameters a and b, declared for functionaddition.

At the point at which the function is called from within main, the control is passed to function addition: here, execution of main is stopped, and will only resume once the addition function ends. At the moment of the function call, the value of both arguments (5 and 3) are copied to the local variables int a and int b within the function. Then, inside addition, another local variable is declared (int r), and by means of the

expression r=a+b, the result of aplus b is assigned to r; which, for this case, where a is 5 and b is 3, means that 8 is assigned to r.

The final statement within the function:

```
return r;
```

Ends function addition, and returns the control back to the point where the function was called; in this case: to functionmain. At this precise moment, the program resumes its course on main returning exactly at the same point at which it was interrupted by the call to addition. But additionally, because addition has a return type, the call is evaluated as having a value, and this value is the value specified in the return statement that ended addition: in this particular case, the value of the local variable r, which at the moment of the return statement had a value of 8.

```
int addition (int a, int b)
   ↓8
z = addition (  5  ,   3  );
```

Therefore, the call to addition is an expression with the value returned by the function, and in this case, that value, 8, is assigned to z. It is as if the entire function call (addition(5,3)) was replaced by the value it returns (i.e., 8).

Then main simply prints this value by calling:

```
cout<< "The result is " << z;
```

A function can actually be called multiple times within a program, and its argument is naturally not limited just to literals:

```
// function example                          The first result is 5
#include <iostream>                          The second result is 5
using namespace std;                         The third result is 2
                                             The fourth result is 6

int subtraction (int a, int b)
{
int r;
```

```
  r=a-b;
  return r;
}


int main ()
{
int x=5, y=3, z;
  z = subtraction (7,2);
cout<< "The first result is " << z << '\n';
cout<< "The second result is " << subtraction (7,2) << '\n';
cout<< "The third result is " << subtraction (x,y) << '\n';
  z= 4 + subtraction (x,y);
cout<< "The fourth result is " << z << '\n';
}
```

Similar to the addition function in the previous example, this example defines
a subtract function, that simply returns the difference between its two parameters. This
time, main calls this function several times, demonstrating more possible ways in which a
function can be called.

Let's examine each of these calls, bearing in mind that each function call is itself an expression
that is evaluated as the value it returns. Again, you can think of it as if the function call was itself
replaced by the returned value:

z = subtraction (7,2);

cout<< "The first result is " << z;

If we replace the function call by the value it returns (i.e., 5), we would have:

z = 5;

cout<< "The first result is " << z;

With the same procedure, we could interpret:

  cout<< "The second result is " << subtraction (7,2);

as:

cout<< "The second result is " << 5;

since 5 is the value returned by subtraction (7,2).

In the case of:

cout<< "The third result is " << subtraction (x,y);

The arguments passed to subtraction are variables instead of literals. That is also valid, and works fine. The function is called with the values x and y have at the moment of the call: 5 and 3 respectively, returning 2 as result.

The fourth call is again similar:

z = 4 + subtraction (x,y);

The only addition being that now the function call is also an operand of an addition operation. Again, the result is the same as if the function call was replaced by its result: 6. Note, that thanks to the commutative property of additions, the above can also be written as:

z = subtraction (x,y) + 4;

With exactly the same result. Note also that the semicolon does not necessarily go after the function call, but, as always, at the end of the whole statement. Again, the logic behind may be easily seen again by replacing the function calls by their returned value:

z = 4 + 2;   // same as z = 4 + subtraction (x,y);

z = 2 + 4;   // same as z = subtraction (x,y) + 4;

Functions with no type. The use of void

The syntax shown above for functions:

type name (argument1, argument2 ...) { statements }

Requires the declaration to begin with a type. This is the type of the value returned by the function. But what if the function does not need to return a value? In this case, the type to be used is void, which is a special type to represent the absence of value. For example, a function that simply prints a message may not need to return any value:

// void function example   I'm a function!

#include <iostream>

using namespace std;

```
void printmessage ()
{
cout<< "I'm a function!";
}
int main ()
{
printmessage ();
}
```

void can also be used in the function's parameter list to explicitly specify that the function takes no actual parameters when called. For example, printmessage could have been declared as:

```
void printmessage (void)
{
cout<< "I'm a function!";
}
```

In C++, an empty parameter list can be used instead of void with same meaning, but the use of void in the argument list was popularized by the C language, where this is a requirement. Something that in no case is optional are the parentheses that follow the function name, neither in its declaration nor when calling it. And even when the function takes no parameters, at least an empty pair of parentheses shall always be appended to the function name. See how printmessage was called in an earlier example:

```
printmessage ();
```

The parentheses are what differentiate functions from other kinds of declarations or statements. The following would not call the function:

```
printmessage;
```

The return value of main

If the execution of main ends normally without encountering a return statement the compiler assumes the function ends with an implicit return statement:

> return 0;

Note that this only applies to function main for historical reasons. All other functions with a return type shall end with a proper return statement that includes a return value, even if this is never used.

When main returns zero (either implicitly or explicitly), it is interpreted by the environment as that the program ended successfully. Other values may be returned by main, and some environments give access to that value to the caller in some way, although this behavior is not required nor necessarily portable between platforms. The values for main that are guaranteed to be interpreted in the same way on all platforms are:

value	description
0	The program was successful
EXIT_SUCCESS	The program was successful (same as above). This value is defined in header <cstdlib>.
EXIT_FAILURE	The program failed. This value is defined in header <cstdlib>.

Because the implicit return 0; statement for main is a tricky exception, some authors consider it good practice to explicitly write the statement.

Command-line parameters are passed to a program at runt-time by the operating system when the program is requested by another program, such as a command interpreter ("shell") like cmd.exe on Windows or bash on Linux and OS X. The user types a command and the shell calls the operating system to run the program.

The uses for command-line parameters are various, but the main two are:

1. Modifying program behaviour - command-line parameters can be used to tell a program how you expect it to behave; for example, some programs have a -q (quiet) option to tell them not to output as much text.

2. Having a program run without user interaction - this is especially useful for programs that are called from scripts or other programs.

The command-line

Adding the ability to parse command-line parameters to a program is very easy. Every C and C++ program has a mainfunction. In a program without the capability to parse its command-line, main is usually defined like this:

int main() Edit & Run

To see the command-line we must add two parameters to main which are, by convention, named argc (**arg**ument **c**ount) and argv (**arg**ument **v**ector [here, vector refers to an array, not a C++ or Euclidean vector]). argc has the type int andargv usually has the type char** or char* [] (see below). main now looks like this:

int main(intargc, char* argv[]) // or char** argv Edit & Run

argc tells you how many command-line arguments there were. It is always at least 1, because the first string in argv(argv[0]) is the command used to invoke the program. argv contains the actual command-line arguments as an array of strings, the first of which (as we have already discovered) is the program's name. Try this example:

```
#include <iostream>

intmain(intargc, char* argv[])
{
std::cout<<argv[0] <<std::endl;
    return 0;
}
```

This program will print the name of the command you used to run it: if you called the executable "a.exe" (Windows) or "a.out" (UNIX) it would likely print "a.exe" or "./a.out" (if you ran it from the shell) respectively.

Earlier it was mentioned that argc contains the number of arguments passed to the program. This is useful as it can tell us when the user hasn't passed the correct number of arguments, and we can then inform the user of how to run our program:

```
#include <iostream>
intmain(intargc, char* argv[])
{
   // Check the number of parameters
   if (argc< 2) {
      // Tell the user how to run the program
std::cerr<< "Usage: " <<argv[0] << " NAME" <<std::endl;
      /* "Usage messages" are a conventional way of telling the user
       * how to run a program if they enter the command incorrectly.
       */
      return 1;
   }
   // Print the user's name:
std::cout<<argv[0] << "says hello, " <<argv[1] << "!" <<std::endl;
   return 0;
}
```

Example output (no arguments passed):

Usage: a.exe <NAME>

Example output (one argument passed):

a.exe says hello, Chris!

Arguments and Parameters

Arguments and parameters are strings passed to your program to give it information. A program for moving files, for example, may be invoked with two arguments - the source file and the destination: move /path/to/source /path/to/destination (note: on Windows these paths would use backslashes instead [and would probably have a drive prefix, like C:], In this example, the program would look something like this:

```
#include <iostream>
intmain(intargc, char* argv[])
{
```

```
   if (argc< 3) { // We expect 3 arguments: the program name, the source path and the
destination path
std::cerr<< "Usage: " <<argv[0] << "SOURCE DESTINATION" <<std::endl;
     return 1;
   }
   return move(argv[1], argv[2]); // Implementation of the move function is platform dependent
}
```

If we wanted to allow the use of multiple source paths we could use a loop and a std::vector:

```
#include <iostream>
#include <string>
#include <vector>
intmain(intargc, char* argv[])
{
   if (argc< 3) { // We expect 3 arguments: the program name, the source path and the
destination path
std::cerr<< "Usage: " <<argv[0] << "SOURCE DESTINATION" <<std::endl;
     return 1;
   }
std::vector <std::string> sources;
std::string destination;
   for (inti = 1; i<argc; ++i) { // Remember argv[0] is the path to the program, we want from
argv[1] onwards
     if (i + 1 <argc)
sources.push_back(argv[i]); // Add all but the last argument to the vector.
     else
       destination = argv[i];
   }
   return move(sources, destination);
```

Arguments may be passed as values to options. An option usually starts with a single hyphen (-) for a "short option" or a double hyphen (--) for a "long option" on UNIX, or a forward slash on Windows. Hyphens (single and double) will be used in this concept. Continuing the example of the move program, the program could use a -d/--destination option to tell it which path is the source and which is the destination, as in move -d /path/to/destination /path/to/source and move --destination /path/to/destination /path/to/source. Options are always right-associative, meaning that the argument to an option is always the text directly to the right of it.

Passing variable number of arguments

When a function is declared, the data-type and number of the passed arguments are usually fixed at compile time. But sometimes we require a function that is able to accept a variable number of arguments. The data-type and/or number of the passed arguments are provided at the run-time.

The secret to passing variable number and type of arguments is the stdarg library. It provides the va_list data-type, which can contain the list of arguments passed into a function. The stdarg library also provides several macros : var_arg, va_start, and va_end that are useful for manipulating the argument-list.

Functions of the macros :

(1) va_start is a macro used to initialize the argument list so that we can begin reading arguments from it. It takes two arguments : (a) the va_list object which stores the passed arguments, and (b) the last named argument, after which the number of arguments is variable.
(2) va_arg is the macro used to read an argument from the list. It takes two parameters: (a) the va_list object we created, and (b) a data type. va_arg returns the next argument as this type.
(3) va_end is a macro that cleans up our va_list object when we're done with it.

<u>**Example 1 :**</u> **A function accepts variable arguments of known data-type**

(A simple average function, that takes variable number of arguments)

```
#include <stdio.h>
#include <stdarg.h>
float avg( int Count, ... )
{
```

```
va_list Numbers;
va_start(Numbers, Count);
int Sum = 0;
for(inti = 0; i< Count; ++i )
        Sum += va_arg(Numbers, int);
va_end(Numbers);
    return (Sum/Count);
}
intmain()
{
    float Average = avg(10, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9);
printf("Average of first 10 whole numbers : %f\n", Average);
    return 0;
}
```

Output of the above code is :

Average of first 10 whole numbers : 4.000000

Example 2 : A function accepts variable arguments of unknown data-type

(A simple print function, that takes variable number and variable type of arguments)

Code:

```
#include <stdio.h>
#include <stdarg.h>
float Print( const char* Format, ... )
{
va_list Arguments;
va_start(Arguments, Format);
    double FArg;
intIArg;
for(inti = 0; Format[i] != '\0'; ++i )
    {
```

```
        if (Format[i] == 'f')

        {
FArg=va_arg(Arguments, double);
printf("Caught a float : %.3lf\n",FArg);

        }
        else if (Format[i] == 'i')

        {
IArg=va_arg(Arguments, int);
printf("Caught an integer : %d\n",IArg);

        }
    }
va_end(Arguments);

}
intmain()
{
Print("This is funny, isn't it ?", 1, 2, 12.1200, 3, 4);
    return 0;
}
```

Output of the above code is :

Caught an integer : 1

Caught an integer : 2

Caught a float : 12.120

Caught an integer : 3

Caught an integer : 4

Arrays

An array is a series of elements of the same type placed in contiguous memory locations that can be individually referenced by adding an index to a unique identifier.

That means that, for example, five values of type int can be declared as an array without having to declare 5 different variables (each with its own identifier). Instead, using an array, the

five int values are stored in contiguous memory locations, and all five can be accessed using the same identifier, with the proper index.

For example, an array containing 5 integer values of type int called foo could be represented as:
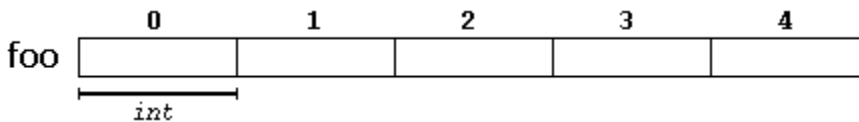
where each blank panel represents an element of the array. In this case, these are values of type int. These elements are numbered from 0 to 4, being 0 the first and 4 the last; In C++, the first element in an array is always numbered with a zero (not a one), no matter its length.

Like a regular variable, an array must be declared before it is used. A typical declaration for an array in C++ is:

type name [elements];

where type is a valid type (such as int, float...), name is a valid identifier and the elements field (which is always enclosed in square brackets []), specifies the length of the array in terms of the number of elements.

Therefore, the foo array, with five elements of type int, can be declared as:

int foo [5];

NOTE: The elements field within square brackets [], representing the number of elements in the array, must be a *constant expression*, since arrays are blocks of static memory whose size must be determined at compile time, before the program runs.

Initializing arrays

By default, regular arrays of *local scope* (for example, those declared within a function) are left uninitialized. This means that none of its elements are set to any particular value; their contents are undetermined at the point the array is declared.

But the elements in an array can be explicitly initialized to specific values when it is declared, by enclosing those initial values in braces {}. For example:

```
int foo [5] = { 16, 2, 77, 40, 12071 };
```

This statement declares an array that can be represented like this:
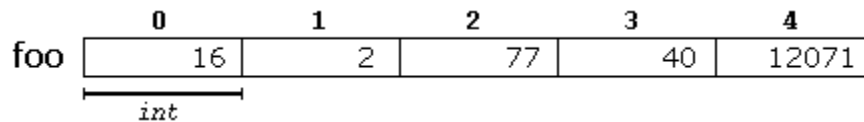
The number of values between braces {} shall not be greater than the number of elements in the array. For example, in the example above, foo was declared having 5 elements (as specified by the number enclosed in square brackets, []), and the braces {} contained exactly 5 values, one for each element. If declared with less, the remaining elements are set to their default values (which for fundamental types, means they are filled with zeroes). For example:

```
int bar [5] = { 10, 20, 30 };
```

Will create an array like this:
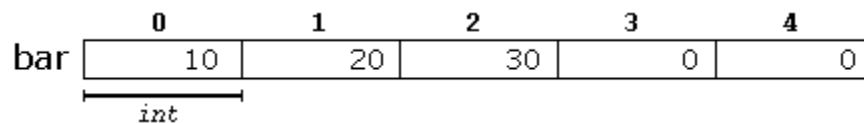
The initializer can even have no values, just the braces:

```
intbaz [5] = { };
```

This creates an array of five int values, each initialized with a value of zero:
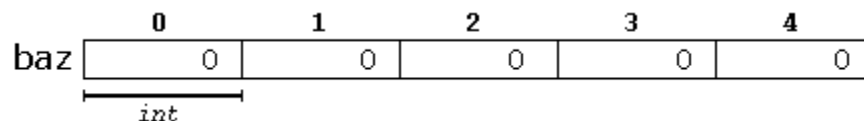
When an initialization of values is provided for an array, C++ allows the possibility of leaving the square brackets empty[]. In this case, the compiler will assume automatically a size for the array that matches the number of values included between the braces {}:

```
int foo [] = { 16, 2, 77, 40, 12071 };
```

After this declaration, array foo would be 5 int long, since we have provided 5 initialization values.

Finally, the evolution of C++ has led to the adoption of *universal initialization* also for arrays. Therefore, there is no longer need for the equal sign between the declaration and the initializer. Both these statements are equivalent:

intfoo[] = { 10, 20, 30 };

int foo[] { 10, 20, 30 };

Static arrays, and those declared directly in a namespace (outside any function), are always initialized. If no explicit initializer is specified, all the elements are default-initialized (with zeroes, for fundamental types).

Accessing the values of an array

The values of any of the elements in an array can be accessed just like the value of a regular variable of the same type. The syntax is:

name[index]

Following the previous examples in which foo had 5 elements and each of those elements was of type int, the name which can be used to refer to each element is the following:
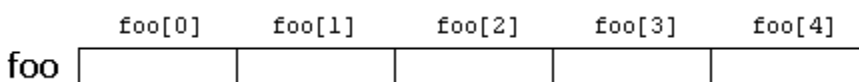
For example, the following statement stores the value 75 in the third element of foo:

foo [2] = 75;

and, for example, the following copies the value of the third element of foo to a variable called x:

x = foo[2];

Therefore, the expression foo[2] is itself a variable of type int.

Notice that the third element of foo is specified foo[2], since the first one is foo[0], the second one is foo[1], and therefore, the third one is foo[2]. By this same reason, its last element

is foo[4]. Therefore, if we write foo[5], we would be accessing the sixth element of foo, and therefore actually exceeding the size of the array.

In C++, it is syntactically correct to exceed the valid range of indices for an array. This can create problems, since accessing out-of-range elements do not cause errors on compilation, but can cause errors on runtime.

At this point, it is important to be able to clearly distinguish between the two uses that brackets [] have related to arrays. They perform two different tasks: one is to specify the size of arrays when they are declared; and the second one is to specify indices for concrete array elements when they are accessed. Do not confuse these two possible uses of brackets []with arrays.

```
intfoo[5];        // declaration of a new array
foo[2] = 75;      // access to an element of the array.
```

The main difference is that the declaration is preceded by the type of the elements, while the access is not.

Some other valid operations with arrays:

```
foo[0] = a;
foo[a] = 75;
b = foo [a+2];
foo[foo[a]] = foo[2] + 5;
```

For example:

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

int foo [] = {16, 2, 77, 40, 12071};
int n, result=0;
```

```
int main ()
{
  for ( n=0 ; n<5 ; ++n )
  {
    result += foo[n];
  }
cout<< result;
  return 0;
}
```

Multidimensional arrays

Multidimensional arrays can be described as "arrays of arrays". For example, a bidimensional array can be imagined as a two-dimensional table made of elements, all of them of a same uniform data type.

jimmy represents a bidimensional array of 3 per 5 elements of type int. The C++ syntax for this is:

int jimmy [3][5];

and, for example, the way to reference the second element vertically and fourth horizontally in an expression would be:

jimmy[1][3]

(remember that array indices always begin with zero).

Multidimensional arrays are not limited to two indices (i.e., two dimensions). They can contain as many indices as needed. Although be careful: the amount of memory needed for an array increases exponentially with each dimension. For example:

char century [100][365][24][60][60];

declares an array with an element of type char for each second in a century. This amounts to more than 3 billion char! So this declaration would consume more than 3 gigabytes of memory!

At the end, multidimensional arrays are just an abstraction for programmers, since the same results can be achieved with a simple array, by multiplying its indices:

int jimmy [3][5];   // is equivalent to

int jimmy [15];   // (3 * 5 = 15)

With the only difference that with multidimensional arrays, the compiler automatically remembers the depth of each imaginary dimension. The following two pieces of code produce the exact same result, but one uses a bidimensional array while the other uses a simple array:

multidimensional array	pseudo-multidimensional array
#define WIDTH 5	#define WIDTH 5
#define HEIGHT 3	#define HEIGHT 3
int jimmy [HEIGHT][WIDTH];	int jimmy [HEIGHT * WIDTH];
intn,m;	intn,m;
int main ()	int main ()
{	{
for (n=0; n<HEIGHT; n++)	for (n=0; n<HEIGHT; n++)
for (m=0; m<WIDTH; m++)	for (m=0; m<WIDTH; m++)
{	{
jimmy[n][m]=(n+1)*(m+1);	jimmy[n*WIDTH+m]=(n+1)*(m+1);
}	}
}	}

None of the two code snippets above produce any output on the screen, but both assign values to the memory block called jimmy in the following way:

	0	**1**	**2**	**3**	**4**
0	1	2	3	4	5
1	2	4	6	8	10
2	3	6	9	12	15

jimmy {

Note that the code uses defined constants for the width and height, instead of using directly their numerical values. This gives the code a better readability, and allows changes in the code to be made easily in one place.

Arrays as parameters

At some point, we may need to pass an array to a function as a parameter. In C++, it is not possible to pass the entire block of memory represented by an array to a function directly as an argument. But what can be passed instead is its address. In practice, this has almost the same effect, and it is a much faster and more efficient operation.

To accept an array as parameter for a function, the parameters can be declared as the array type, but with empty brackets, omitting the actual size of the array. For example:

```
void procedure (intarg[])
```

This function accepts a parameter of type "array of int" called arg. In order to pass to this function an array declared as:

```
intmyarray [40];
```

it would be enough to write a call like this:

```
procedure (myarray);
```

Here you have a complete example:

```
// arrays as parameters                    5 10 15
#include <iostream>                         2 4 6 8 10
```

using namespace std;

```
void printarray (intarg[], int length) {
  for (int n=0; n<length; ++n)
cout<<arg[n] << ' ';
cout<< '\n';
}
```

```
int main ()
{
intfirstarray[] = {5, 10, 15};
intsecondarray[] = {2, 4, 6, 8, 10};
printarray (firstarray,3);
printarray (secondarray,5);
}
```

In the code above, the first parameter (intarg[]) accepts any array whose elements are of type int, whatever its length. For that reason, we have included a second parameter that tells the function the length of each array that we pass to it as its first parameter. This allows the for loop that prints out the array to know the range to iterate in the array passed, without going out of range.

In a function declaration, it is also possible to include multidimensional arrays. The format for a tridimensional array parameter is:

```
base_type[][depth][depth]
```

For example, a function with a multidimensional array as argument could be:

```
void procedure (intmyarray[][3][4])
```

Notice that the first brackets [] are left empty, while the following ones specify sizes for their respective dimensions. This is necessary in order for the compiler to be able to determine the depth of each additional dimension.

In a way, passing an array as argument always loses a dimension. The reason behind is that, for historical reasons, arrays cannot be directly copied, and thus what is really passed is a pointer. This is a common source of errors for novice programmers.

Manipulating array elements using loops

Arrays and loops

One of the nice things about arrays is that you can use a loop to manipulate each element. When an array is declared, the values of each element are not set to zero automatically.

In some cases **you** want to "re-initialize" the array (which means, setting every element to zero). This can be done like in the example above, but it is easier to use a loop. Here is an example:

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

intmain()
{
        inta[4];
        inti;

        for ( i = 0; i< 4; i++ )
                a[i] = 0;
        for ( i = 0; i< 4; i++ )
                cout<< a[i] << '\n';
        return 0;
}
```

Note: In the first "for loop" all elements are set to zero. The second "for loop" will print each element.

intarr[10];//array of integers input by user

intnum;   //smallest number in array

```
inttemp;   //temp variable for swapping numbers

intind;    //index of where temp was found

cout<< "Enter ten random integers: " <<endl;

for(inti=0; i<10; i++)

{

cout<< "[" <<i<< "] = ";

cin>>arr[i];

}


cout<<endl;

for (int j=0; j<10; j++)

{

num = arr[j];

   temp = arr[j];


   for (int k=j; k<10; k++) /*after this loop, temp should have lowest int and ind

                   should have its location*/

   {

if(temp >arr[k])

    {

       temp = arr[k];

ind = k;

    }

   }

arr[j] = temp;

arr[ind] = num;

}


for(int l = 0; l<10; l++)

{
```

```
cout<<arr[l] << " ";
    }
```

Use Various types of arrays (integer, float and character arrays / Strings)

Integer array

C++ Program to store 5 numbers entered by user in an array and display first and last number only.

```
#include <iostream>
using namespace std;
intmain() {
intn[5];
cout<<"Enter 5 numbers: ";
/*  Storing 5 number entered by user in an array using for loop. */
    for (inti = 0; i< 5; ++i) {
cin>>n[i];
    }
cout<<"First number: "<<n[0]<<endl;  // first element of an array is n[0]
cout<<"Last number: "<<n[4];       // last element of an array is n[SIZE_OF_ARRAY - 1]
    return 0;
}
```

Output

Enter 5 numbers: 4

-3

5

2

0

First number: 4

Last number: 0

Float array0

```
#include<iostream>
usingnamespacestd;
intmain(){
constintMAX_STUDENTS=4;
floatstudentGrades[ MAX_STUDENTS ]={0.0};
for(inti=0;i<MAX_STUDENTS;i++){
cout<<i<<" "<<studentGrades[i]<<'\n';
}
return0;
}
```

The program gave the expected results:

00

10

20

30

Character array

Character sequences

The string class is a very powerful class to handle and manipulate strings of characters. However, because strings are, in fact, sequences of characters, we can represent them also as plain arrays of elements of a character type.

For example, the following array:

char foo [20];

is an array that can store up to 20 elements of type char. It can be represented as:

foo
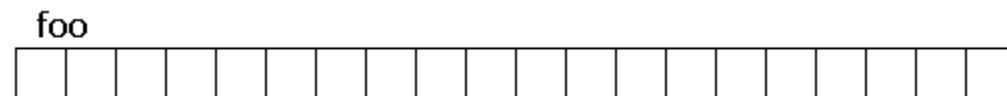
Therefore, this array has a capacity to store sequences of up to 20 characters. But this capacity does not need to be fully exhausted: the array can also accommodate shorter sequences. For example, at some point in a program, either the sequence "Hello" or the sequence "Merry Christmas" can be stored in foo, since both would fit in a sequence with a capacity for 20 characters.

By convention, the end of strings represented in character sequences is signaled by a special character: the *null character*, whose literal value can be written as '\0' (backslash, zero).

In this case, the array of 20 elements of type char called foo can be represented storing the character sequences "Hello"and "Merry Christmas" as:

foo

| H | e | l | l | o | \0 | | | | | | | | | | | | | | |

| M | e | r | r | y | | C | h | r | i | s | t | m | a | s | \0 | | | | |

Notice how after the content of the string itself, a null character ('\0') has been added in order to indicate the end of the sequence. The panels in gray color represent char elements with undetermined values.

Initialization of null-terminated character sequences

Because arrays of characters are ordinary arrays, they follow the same rules as these. For example, to initialize an array of characters with some predetermined sequence of characters, we can do it just like any other array:

```
charmyword[] = { 'H', 'e', 'l', 'l', 'o', '\0' };
```

The above declares an array of 6 elements of type char initialized with the characters that form the word "Hello" plus a*null character* '\0' at the end.

But arrays of character elements have another way to be initialized: using *string literals* directly. string literals have already shown up several times. These are specified by enclosing the text between double quotes ("). For example:

```
"the result is: "
```

This is a *string literal*, probably used in some earlier example.

Sequences of characters enclosed in double-quotes (") are *literal constants*. And their type is, in fact, a null-terminated array of characters. This means that string literals always have a null character ('\0') automatically appended at the end.

Therefore, the array of char elements called myword can be initialized with a null-terminated sequence of characters by either one of these two statements:

```
1 charmyword[] = { 'H', 'e', 'l', 'l', 'o', '\0' };
2 charmyword[] = "Hello";
```

In both cases, the array of characters myword is declared with a size of 6 elements of type char: the 5 characters that compose the word "Hello", plus a final null character ('\0'), which specifies the end of the sequence and that, in the second case, when using double quotes (") it is appended automatically.

Please notice that here we are talking about initializing an array of characters at the moment it is being declared, and not about assigning values to them later (once they have already been declared). In fact, because string literals are regular arrays, they have the same restrictions as these, and cannot be assigned values.

Expressions (once myword has already been declared as above), such as:

```
1 myword = "Bye";
2 myword[] = "Bye";
```

would **not** be valid, like neither would be:

```
myword = { 'B', 'y', 'e', '\0' };
```

This is because arrays cannot be assigned values. Note, though, that each of its elements can be assigned a value individually. For example, this would be correct:

```
1 myword[0] = 'B';
2 myword[1] = 'y';
```

```
3 myword[2] = 'e';
4 myword[3] = '\0';
```

Strings and null-terminated character sequences

Plain arrays with null-terminated sequences of characters are the typical types used in the C language to represent strings (that is why they are also known as *C-strings*). In C++, even though the standard library defines a specific type for strings (class string), still, plain arrays with null-terminated sequences of characters (C-strings) are a natural way of representing strings in the language; in fact, string literals still always produce null-terminated character sequences, and not string objects.

In the standard library, both representations for strings (C-strings and library strings) coexist, and most functions requiring strings are overloaded to support both.

For example, cin and cout support null-terminated sequences directly, allowing them to be directly extracted from cin or inserted into cout, just like strings. For example:

```
// strings and NTCS:                         What is your name? Homer
#include <iostream>                          Where do you live? Greece
#include <string>                            Hello, Homer from Greece!
usingnamespacestd;
int main (){
char question1[] = "What is your name? ";
  string question2 = "Where do you live? ";
char answer1 [80];
  string answer2;
cout<< question1;
cin>> answer1;
cout<< question2;
cin>> answer2;
```

cout<<"Hello, "<< answer1;

cout<<" from "<< answer2 <<"!\n";

return 0;

}

In this example, both arrays of characters using null-terminated sequences and strings are used. They are quite interchangeable in their use together with cin and cout, but there is a notable difference in their declarations: arrays have a fixed size that needs to be specified either implicit or explicitly when declared; question1 has a size of exactly 20 characters (including the terminating null-characters) and answer1 has a size of 80 characters; while strings are simply strings, no size is specified. This is due to the fact that strings have a dynamic size determined during runtime, while the size of arrays is determined on compilation, before the program runs.

In any case, null-terminated character sequences and strings are easily transformed from one another:

Null-terminated character sequences can be transformed into strings implicitly, and strings can be transformed into null-terminated character sequences by using either of string's member functions c_str or data:

```
1 charmyntcs[] = "some text";
2 string mystring = myntcs;  // convert c-string to string
3 cout<<mystring;   // printed as a library string
4 cout<<mystring.c_str();  // printed as a c-string
```

Two-Dimensional Arrays

A 2-dimensional array is an array of arrays. In other words, it is an array where each member of the array is also an array. Consider the following table

Country\Data	Map	Flag	Area (sq km)	Population
United States			9,629,091	272,639,608

Cameroon			475,440	15,456,092
Guatemala			108,890	12,335,580
Italy			301,230	56,735,130
Oman			212,460	2,446,645

Declaring and Initializing a 2-Dimensional Array

This two-dimensional array is made of rows and columns . Each column represents one category of data that everyone of the rows shares with the other rows. As different as each map looks, it still remains a map; each country on the table is known for its map, its flag, itsarea, and its population, though remaining different from the others. To see another two-dimensional array, look at a calendar that displays a month with its week days.

Like the above table, a 2-dimensional array is made rows and columns. To declare it, usedouble pair of a opening and closing square brackets. Here is an example:

 intnumberOfStudentsPerClass[12][50];

This declaration creates a first group of 12 elements; it could be an array of 12 classes. Each element of the array contains 50 elements. In other words, each of the 12 members of the group is an array of 50 items. Simply stated, this declarations creates 12 classes and each class contains 50 students.

Before using the members of an arrays, you should/must make sure you know the values that its members hold. As done with one-dimensional arrays, there are two ways you can solve this problem: you can initialize the array or you can get its values by another means.

You can initialize an array the same way you would proceed the a one-dimensional array: simply provide a list of values in the curly brackets. A multidimensional array is represented as an algebraic matrix as MxN. This means that the array is made of M rows and N columns. For example, a 5x8 matrix is made of 5 rows and 8 columns. To know the actual number of members of a multidimensional array, you can multiply the number of rows by the number of columns.

Therefore a 2x16 array contains 2*16=32 members.

Based on this, when initializing a 2-dimensional array, make sure you provide a number of values that is less than or equal to the number of members.

Here is an example:

double distance[2][4] = {44.14, 720.52, 96.08, 468.78, 6.28, 68.04, 364.55, 6234.12};

To locate a member of the array, this time, each must be identified by its double index. The first member is indexed at [0][0]. The second is at [0][1]. For a 2x4 array as this one, the 5th member is at [1][0]. You can use this same approach to display the values of the members of the array. Here is an example:

```cpp
#include <iostream>
using namespace std;
intmain()
{
        // A 2-Dimensional array
        double distance[2][4] = {44.14, 720.52, 96.08, 468.78, 6.28, 68.04, 364.55, 6234.12};

        // Scan the array from the 3rd to the 7th member
        cout<< "Members of the array";
        cout<< "\nDistance [0][0]" << ": " <<distance[0][0];
        cout<< "\nDistance [0][1]" << ": " <<distance[0][1];
        cout<< "\nDistance [0][2]" << ": " <<distance[0][2];
        cout<< "\nDistance [0][3]" << ": " <<distance[0][3];
        cout<< "\nDistance [1][0]" << ": " <<distance[1][0];
        cout<< "\nDistance [1][1]" << ": " <<distance[1][1];
        cout<< "\nDistance [1][2]" << ": " <<distance[1][2];
        cout<< "\nDistance [1][3]" << ": " <<distance[1][3];
        cout<<endl;
        return 0;
}
```

This would produce:

Members of the array

Distance [0][0]: 44.14

Distance [0][1]: 720.52

Distance [0][2]: 96.08

Distance [0][3]: 468.78

Distance [1][0]: 6.28

Distance [1][1]: 68.04

Distance [1][2]: 364.55

Distance [1][3]: 6234.12

To make the above array a little easier to read when initializing it, you can type the values of each row on its own line. For example, the above array can be initialized as follows:

double distance[2][4] = { 44.14, 720.52, 96.08, 468.78,

6.28, 68.04, 364.55, 6234.12 };

C++ also allows you to include each row in its own pair of curly brackets. You must separate each row from the next with a comma. Once again, this makes code easier to read. Here is an example:

double distance[2][4] = { { 44.14, 720.52, 96.08, 468.78 },

{ 6.28, 68.04, 364.55, 6234.12 }

};

POSSIBLE QUESTIONS – UNIT II
Part-A
Online Examinations   (One marks)

1. _____ inherits get(), getline(), read(), seekg(), and tellg() from istream.

    a) conio   **b) ifstream**   c) fstream   d) iostream

2. What function should be used to free the memory allocated by calloc() ?

    a) dealloc();   b) malloc(variable_name, 0)

    c) free();   d) memalloc(variable_name, 0)

3. The _____ is special because its name is the same as the class name.

    a) Destructor   b) static   **c) constructor**   d) none

4. The class _____ describes how the class function are implemented

    a) Function definition   b) declaration   c) arguments   d) none

5. A derived class with only one base class is called _____ inheritance.

    a) Single   b) Multi-level   c) Multiple   d) Hierarchical

6. The exception handling mechanism is basically built upon _____ keyword

    a) try   b) catch   c) throw   **d) all the above**

7. _____ functions must either be member functions or friend functions.

    a) Operator   b) User-defined   c) Static Member   d) Overloading

8. Which is more effective while calling the functions?

    a) call by value   **b) call by reference**   c) call by pointer   d) none

9. Where the default values of parameter have to be specified?

    a) Function call   b) Function definition   **c) Function prototype**   d) Both B or C

10. Which is more memory efficient?

    a) structure   **b) union**

    c) both use same memory   d) depends on a programmer

Part-B - 2 MARKS

1. Define void functions.

2. Define inline function with example.

3. What is array write its types?

4. What is multidimensional array with example?

5. What are functions?

6. Define call by value.

7. Define call by reference.

8. What is a function? How will you define a function?

Part-C 6 MARKS

1. Define Functions. Explain call by value with suitable example program.

2. Explicate types of arrays. Explain One Dimensional array with program.

3. Define Functions. Explain call by reference with suitable example program.

4. List out different types of arrays. Explain Multi Dimensional array with program

5. Write note on   i) void functions   ii) Function Parameters

6. Explain Command Line arguments with suitable example program

7. Explain Inline functions with suitable example program.

8. Write notes on i) Call by value ii)Call by reference of functions.

9. Explain in detail about String functions with syntax and example.

10. Write the syntax for declaring and initializing a multidimensional array. Give example.

11. Write in detail about functions that pass variable number of arguments. Explain with syntax and example.

12. How will you declare and initialize a two dimensional array? Write a C program to perform matrix addition.

13. What are functions? Explain the various categories of functions with syntax and example.

14. How will you declare and initialize a one dimensional array? Give an example program to assign marks of a student in an array.

Part -A Online Examinations   **(1 mark questions)**
SUBJECT: PROGRAMMING FUNDAMENTALS USING C/C++   **SUBJECT CODE: 17CSU101**

UNIT-II

S.No	Questions	OPT1	OPT2	OPT3	OPT4	Answer
1	A member function can call another member function directly without using the _____ operator	Assignment	equal	dot	greater than	dot
2	A _____ member variable is initialized to zero when the first object of its class is created	Dynamic	constant	static	protected	static
3	_____ Variables are normally used to maintain values common to the entire class.	Private	protected	Public	static	static
4	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
5	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
6	A _____ function can be invoked like a normal function without the help of any object	Void	friend	inline	none of the above	friend

#	Question	A	B	C	D	Answer
7	The _____ member variables must be defined outside the class.	Static	private	public	protected	Static
8	A friend function, although not a member function, has full access right to the _____ members of the class	Static	private	public	protected	private
9	Function should return a _____.	value	character	both (a) and (b)	none	value
10	_____function is useful when calling function is small	Built-in	Inline	user-defined	none.	Inline
11	c++ propouse a new future called _____	function overloading	polymorphism	Inline function	calling function	Inline function
12	Which of the following cannot be passed to a function?	reference variables	arrays	class objects	header files	header files
13	Function should return a _____.	value	character	both (a) and (b)	none	value
14	_____function is useful when calling function is small	Built-in	Inline	user-defined	none.	Inline
15	Inline function needs more_____	variables	functions	memoryspace	control structures	memoryspace
16	Multiple function with the same name is known as _____	function overloading	Encapsulation	inheritance	operator overloading	function overloading
17	The _____ function creates a new set of variables and copies the values of arguments into them.	calling function	called function	function	function overloading	called function

#	Question					
18	Function contained within a class is called a _____	built-in	member function	user-defined function	calling function	member function
19	In c++,Declarations can appear_____in the body of the function	Only at the top	middle	bottom	anywhere	anywhere
20	Modular structure of C language enables the program to be spli	structure	union	integers	function	function
21	The actual and formal arguments of functions must match in _	actual argur	formal arg	dummy pa	temporary variab	actual arguments
22	Functions receives the values passed by the calling function an	actual argun	formal arg	dummy pa	temporary variab	formal arguments
23	In looping process first step is _____	intialise cou	test for cor	increment	execution stateme	intialise counter
24	In case of for loop _____ section is executed before test condi	increment	intialise	testing	execution of state	increment
25	_____ statement is the mechanism for returning value to the ca	return	continue	break	goto	return
26	A function can return ____ value per call	one	zero	two	multiple	one
27	_____ is a special case where a function calls itself.	recursion	subroutine	structure	none	recursion
28	___ is a group of related data items that share a common name.	variables	array	function	structure	array

#	Question					
29	A _____ is an array of characters.	string	variables	function	none	string
30	Individual values in array is referred as _____.	subscript	elements	subelement	none	elements
31	Any subscript between _____ are valid for an array of fifty e	0-49	0-56	0-48	0-46	0-49
32	Value in a matrix can be represented by _____ subscript.	1	3	2	4	2
33	arrays that do not have their dimensions explicitly specified are	unsized arra	undimensio	initialized a	to size arrays	unsized arrays
34	In ASCII character set the uppercase alphabet represent codes _	65 to 90	96 to 45	97 to 123	1 to 26	65 to 90
35	Modular structure of C language enables the program to be spli	structure	union	integers	function	function
36	The actual and formal arguments of functions must match in _	actual arguments	formal arguments	dummy parameters	temporary variables	actual arguments
37	Functions receives the values passed by the calling function an	actual arguments	formal arguments	dummy parameters	temporary variables	formal arguments
38	Process of calling a function using pointers to pass address of	call by value	call by reference	call by method	call by address	call by reference
39	The process of passing actual values of variable is known as __	call by value	call by reference	call by method	call by address	call by value

		values	addresses	operators	none of the above	addresses
40	In pointers when function is called _____ are passed as actual	values	addresses	operators	none of the above	addresses

KARPAGAM ACADEMY OF HIGHER EDUCATION

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

Coimbatore - 641021.

(For the candidates admitted from 2017 onwards)

DEPARTMENT OF COMPUTER SCIENCE,CA & IT

SUBJECT	: **PROGRAMMING FUNDAMENTALS USING C/C++**			
SEMESTER	: **I**			**L T P C**
SUBJECT CODE: 17CSU101	**CLASS**	: **I B.Sc.CS**		**4 0 0 4**

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.

Suggested Readings:

1. Herbtz Schildt, 2003,*C++: The Complete Reference*, 4th Edition, McGraw Hill.

2. Bjarne Stroustrup, 2013, *The C++ Programming Language,* 4th Edition, Addison-Wesley.

3. Bjarne Stroustroup, 2014, *Programming - Principles and Practice using C++,* 2nd Edition, Addison-Wesley.

4. E Balaguruswamy, 2008,*Object Oriented Programming with C++,* 2nd Edition ,Tata McGraw-Hill Education.

5. Paul Deitel, Harvey Deitel, 2011,*C++ How to Program*, 8th Edition, Prentice Hall.

6. John R. Hubbard, 2000, *Programming with C++,* 2nd Edition ,Schaum's Series.

7. Stefan Bjornander, 2016,*C++ Windows Programming,* Published by Packt Publishing Ltd.

8. Scott Meyers, 2005, *Effective C++,* 3rd Edition, Published by Addison-Wesley.

9. Harry, H. Chaudhary, 2014 ,*Head First C++ Programming: The Definitive Beginner's Guide*, First Create space Inc, 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*, 5th Edition, Published by Addison-Wesley.

12. Debasish Jana , 2014,*C++ And Object-Oriented Programming Paradigm*,Published by PHI Learning Pvt. Ltd.

13. Richard L. Stegman, 2016, *Focus on Object-oriented Programming With C++,*6th Edition ,CreateSpace Independent Publishing Platform,.

14. Andrew Koeni, Barbara, E. Moo,2000,*Accelerated using C++,* Published by Addison-Wesley .

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. http://*www.cplusplus.com/*

6. http://*www.cppreference.com/*

UNIT III

Structures

Introduction - What is structure ?

Array is a collection of data items and all data item must be of same type. In very large applications, some data items may be related to others or group of data items may be related. Let us consider the college and the information related to the student such as name; roll number, age, marks etc are heterogeneous types. These items can't be grouped using array. To group these kinds of data items, another feature of C called structure can be used. So "structure" means that related data items may be grouped under same name. By grouping of related items under one name called structure name, we could write programs well.

For example, the information about an employee like employee name, department, designation, salary details can be grouped by one name like emp_record.

How to declare the structure:

Structure declaration is different to the conventional declarations, look the following.

```
struct   <tag>
   {
      member-1;
      member-2;
      ...
      member-n;
   };
```

```
struct  [<tag>]
   {
     member-1;
      member-2;
        ...
      member-n;
   }sv1,sv2...;
```

⇨ struct is a keyword to indicate that structure variable.

⇨ member-1, member-2 are the variables of the structure.

⇨ In the first option the **<tag>** name is must because using this **<tag>** only we can create new structure variable as follows. The structure variable is defined as following format only.

struct <tag> sv1,sv2...;

This declaration is just like as **int a,b,c …;** In the second option **sv1,sv2** are the structure variables. The structure ends with semicolon like others.

The information about students such as name, roll number and marks are grouped and declared as follows.

struct stu

{ char name[16];

int rollno, marks;

```
            };
        struct  stu  s1,s2;
```

Here using the tag name **stu** the structure variable **s1, s2** are created. Alternative way to declare the structure variable is as follows.

```
    struct  stu
    {     char  name[16];
        int   rollno, marks;
    }s1,s2;
```

Now without using tag name the variables **s1**, **s2** are created. So we can use any one of the above declarations.

Referring the data in structure:

The aim of the array and structure is basically same. In array the elements are referred by specifying array name with index like a[5] to refer the fifth element. But in structure, the members are referred by entirely new method as mentioned below.

```
    structure-name. variable name;
```

The dot (**.**) operator is used to refer the members of the structures. For example if we wish to access the members of the previous structure, the following procedure have to be followed.

    s1.name, s1.marks, s1. rollno

Assigning the values to the structure variable:

The values may be assigned for the array while declaring it as follows

                int a[5] = {10,20,30,40,50};

As above we can assign the value for the members of the structure as follows.

```
    struct  stu
    {   char name[15];
        int  rollno, marks;
    } s1= {"Karthi",1000,76};
```

Here the string value **"Karthi"** will be assigned to the member **name**, **1000** will be assigned to the member **rollno** and   **76** will be assigned to **marks**. The following program is a first one using structure and sees how the members of the structure are being referred.

```
/* Example for structure reference and assignment */
```

```
main( )
{  struct stu
      {char name[15];
            int rollno, marks;
      }s1 = {"Karthi",1000,76};
      printf("\nName   = %s ",s1.name);
   printf("\nroll no  = %d ",s1.rollno);
   printf("\nMarks   = %d ",s1.marks);    }
```

Suppose all the values of one structure are necessary for another structure variable. The values can be copied one by one as usual. Here structure supports the whole structure can be assigned using = operator. Assume **s1** and **s2** are the structure variables and the contents of **s1** should be copied in **s2** also. How?

strcpy(s2.name, s1.name);

s2.rollno = s1. rollno; /* copying one by one*/

s2.marks = s1. marks.

(or)

s2 = s1;

 /* Copying entire structure to another structure */

The second one is the best way of programming approach to copying structures. An example program to prepares a pay slip for the employee using the structure.

```
/*To find the net pay of the employee using structure */
main( )
{ struct emp
    { char name[25];
     float bp,hra,pf,da,np;
     int empno;
   }e;
    printf("\nName of the employee : ");
    gets(e.name);
    printf("\nEmployee No : ");
    scanf("%d",&e.empno);
    printf("\nBasic Pay  : ");
    scanf("%f",&e.bp);
    if (e.bp>5000)
    { e.da = 1.25 * e.bp;        /* 125 %  DA */
     e.hra = .25  * e.bp;         /* 25 %  HRA */
     e.pf = .12  * e.bp;         /* 12 %  PF */
    }
  else
  {    e.da = 1.0 * e.bp;  /* 100 % DA */
        e.hra = .15 * e.bp;  /* 15 % HRA */
        e.pf = .10 * e.bp;  /* 10 % PF */
  }
  e.np = e.bp + e.da + e.hra - e.pf;
  printf("\n\tKarthik  Systems pvt. ltd., \n");
  printf("\nName : %s  Employee No :  %d \n",e.name,e.empno);
  printf("\nBasic Pay  D.A     H.R.A    P.F Net Pay\n");
  printf("\n%5.2f    %5.2f  %5.2f  %5.2f  %5.2f ",e.bp, e.da, e.hra, e.pf,
e.np);
  }
```

Array of structures:

The above example is only for manipulating single record, that is only one employee information. Suppose if we want to prepare more number of records, we can use the array of structures. Array of structure is defined as simple as ordinary arrays as below

struct emp e[100];

The above declaration indicates that **e** is a array of structure variable and we can store **100** employees information. The reference of members is also similar to the array reference. So, first we have to specify the index of the structure and necessary variables. To refer the first employee's information we have to use the notations

s[0]**.**name, s[0]**.**np etc.

Like wise all the employees information are referred and processed. The following example illustrates the array of structures.

```
/* To find the class of the students  */
main()
{        struct stu
         {
          char name[25];
          int rollno,marks;
         }s[50];
         int n,i;
         char result[15];
         printf("\nHow many students  : ");
         scanf("%d",&n);
         printf("\nEnter %d  students information\n",n);
         for(i=0;i<n;i++)
         {  printf("\nEnter %d persons name : ",i+1);
          scanf("%s",s[i].name);
          printf("\nRoll No : ");
          scanf("%d",&s[i].rollno);
          printf("\nMarks   : ");
          scanf("%d",&s[i].marks);
         }
```

```
            printf("\nResult of the students ");
            for(i=0;i<n;i++)
             {   if (s[i].marks >= 60)
                 strcpy(result,"First");
               if ((s[i].marks >= 50)  && (s[i].marks <60))
                 strcpy(result,"Second");
              if ((s[i].marks >= 40) && (s[i].marks<50))
                 strcpy(result,"Third");
               if (s[i].marks < 40)
                      strcpy(result,"Fail");
            printf("\nResult = %s class ",result);
     }      }
```

As we know that the elements of array are stored continuously. In structure also the members of structure will be stored in consecutive memory locations one after another. This is illustrated in the following program. It has a structure **stu** and size of single structure is **17** bytes. (**2** for age and **15** for name, so **2+15=17** bytes)

```
    /* Array of structures  */
    struct
    {
      int  age;
      char name[15];
    }stu[5];
    main()
    {
      int i;
      for(i=0;i<5;i++)
        printf("\nAddress is : ",&stu[i]);
}
    Address is : 1200
    Address is : 1217
    Address is : 1234
    Address is : 1251
```

Address is : 1268

From this output it is found that the elements in structure are also stored in consecutive memory locations.

Nested Structure

In case of nested **if,** the statement part will have another **if** statement. In case of nested looping also, the statement portion has another looping statement. So the nested structure also will have another structure variable as a member.

There is no data type for maintaining date related information. Now we are going to create a user defined data type using structure and it can be used as a data type for date.

struct

{   int dd,mm,yy;

}d;

struct

{char name[15];

        struct d   dob;

}stu;

The first structure **d** has three fields to represent a date by using three variables, dd (day), mm(month) and yy(year). The second structure **stu** have two member fields. They are **name** and date of birth (**dob**), which is declared using the structure **d.**

We have an idea about the reference of values of the structure variable. Here to access the **name** is very simple and to access the **dob** is differ. The **dob** structure members are accessed by as follows.

stu**.d.**dd ,   stu**.d.**dd and stu**.d.**dd

To refer the member **dob** we can simply specify **stu . dob** is enough. But **dob** is not an ordinary variable, which is another structure variable with three members. If we made any reference is directly via **dob**, we can refer by **dob.membres**. But if reference is through the **stu**, we have to use the **stu.d.dd** etc., A complete program for nested structure is given below.

/* Example for Nested structure */

struct dob

{

  int dd,mm,yy;

```
            };
        struct
        {
          char name[15];
          struct dob db;
        }stu;
        main()
        {
          clrscr();
          printf("\eEnter the name :");
          scanf("%s",stu.name);
          printf("\nEnter the age (dd/mm/yy) :");
          scanf("%d%d%d",&stu.db.dd,&stu.db.mm,&stu.db.yy);
          printf("\nYour Name is : %s ",stu.name);
          printf("\nDate of Birth : %2d-%2d-%2d",
                      stu.db.dd,stu.db.mm,stu.db.yy);
}
```

```
        Enter the name : Karthi
        Enter the age  (dd/mm/yy)  : 3 4 1974
        Your name is  : Karthi
        Date of Birth : 3-4-1974
```

Structures and functions:

Passing and returning various parameters and returning various values etc also given. Now let us see, how the functions are used in structures also. In a simple function call, we have to mention the name of the function with necessary parameters as given below to pass one integer argument.

void display(int a);

Now we need to pass the structure to the function. What shall we do? One solution is passing the members of structure one by one. But it is not an optimal when there are large members in a structure. Otherwise look the following

void display(struct stu s)

```
/* Passing structure to the function  */
struct stu       /* structure is declared as global  */
```

```
        {        char name[25];
                 int rollno;
        };
        main( )
        { struct stu s1;  /*s1 is only local to main( )  */
         printf("\nName of the student  : ");
         scanf("%s",s1.name);
         printf("\nRoll No.          : ");
         scanf("%d",&s1.rollno);
           display(s1); /* Calling function using structure variable */      }
        /* Structure stu must declared as global otherwise   we can't use this name as in the
        following parameter  declaration */
        void display( struct stu s2)
        {        printf("\nYour  information is \n");
                 printf("\nName    : %s ",s2.name);
        printf("\nRoll No : %d ",s2.rollno);      }
```

Miscellaneous of Structures:

The structure is used to store different type of   values and all the variables are stored continuously as in the following diagram and program illustrates this.

```
        /* Additional to structure */
      main( ){    struct     {      int a, b;    } test;
   printf("\nBase address of structure  : %u ",&test);
   printf("\nAddress of first member 'a': %u ",&test.a);
   printf("\nAddress of second member 'b' : %u ",&test.b);
   printf("\nSize of the structure 'test' : %d bytes ",sizeof(test));  }
```

Base address of structure   : 3354

Address of first   member 'a'   : 3354 /* 2 bytes for int */

Address of second member 'b' : 3356

Size of the structure 'test' : 4 bytes   /* So 2+2=4 bytes */

Starting address of the structure **test** is **3354.** The address of the first structure variable **a** is also same i.e. **3354**. The next variable **b** is stored in the next memory location. (i.e. **3354 + 2 = 3356**

, **int**eger needs **2** bytes memory) The size of the structure is **4** bytes, because of two integer variables (**2** + **2** = **4** bytes).

The memory representation of array of structures is also same for simple arrays and it will be cleared in the following program.

```
/* Array of structure */
main( )
{   struct
    {  char name[16];
       int rollno,marks;
    }s[5];        /* five structures */
int i;
for(i=0;i<5;i++)  printf("\nAddress of structure S[%1d]= %d ",i,&s[i]);
    printf("\nSize of the entire structure = %d  bytes",sizeof(s));
```

         Address of structure S[0] = 8650
         Address of structure S[1] = 8670
         Address of structure S[2] = 8690
         Address of structure S[3] = 8710
         Address of structure S[4] = 8730
         Size of the entire structure = 100 bytes

In the above program the structure **s** is declared as array of structure with the size **5**.

⇨ Address of first structure (**s[0]**) is **8650** and next is at **8670** etc.

⇨ Size of the single structure is **20** bytes (**16** + **2** + **2** = **20**).

⇨ So for **5** structures **100** bytes were needed.

1st structure	2nd structure	3rd structure	4th structure	5th structure	…..

Here each structure occupies **20** bytes of memory and single structure is stored in the memory as follows.

name (16 bytes)	rollno (2 bytes)	marks (2 bytes)

UNIONS

Union is the best gift for the C programmers. Yes. For looking and the general declaration of union is similar to the structure variable. Instead of the key word **struct**, the key word **union** is used. The members of **union** also referred with the help of (.) dot operator.

The union variable has been mainly used to set/reset the status of the hardware, devices of the computer system and its roll is very much in the system software development.

For example, the register has **16** bit and they are named as low byte and high byte. If any changes in the low or high byte will affect the full word of the register.

Difference between structure and union:

In case of structure all the members occupies different memory locations depends on the type, which it belongs to. In union memory will be allocated only for the larger size variable of the group, no other memory allocation will be made. Now, allocated highest memory will be shared by all the remaining variables of the union. The declaration of a union and its format is as follows

General format: Example:

```
union
 {
     member-1;
     member-2;
     member-3;
     ...
     member-n;
  }union-variable;
```

```
union
 {
    char  name[15];
    int   rollno;
    float marks;
 } stu;
```

We may think that the size of the union variable is **21** bytes **(15+2+4)**. But it is not correct. Because of union larger memory request only considered for allocation. No independent memory for the members will be allocated.

```
/* Example for the union variable */
main( )
{union
        { char name[15];
         int  rollno;
         float marks;
        } s;
printf("\nSize of the union  : %d ",sizeof(s));      }
```

Size of the union : 15

In this program the maximum memory request is **15** (char name [15]). So all the remaining members of the union **rollno, marks** will share the same memory area.

Let us consider a union variable with two members one is **int** and another one is **float.** In general **int**eger requires 2 bytes and **float** requires 4 bytes. But in union only the memory for **float** will be allotted and this is also shared by **int** variable also. This discussion is illustrated in the following diagrams.
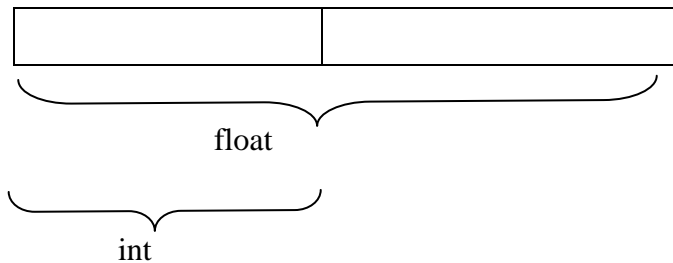
Memory is shared- a proof

The following program illustrates our discussion of previous paragraph idea. The largest memory area will be shared by the other members. If so what is going to happen when we refer. Yes. Confusion. But be clear that two values will be accessed and changes in one disturb the other one.

```
/* A proof of union - sharing memory */
main( )
{     union
     { char c;
       int  a;
     }s;
        s.c= 'z';
       printf("\nC = %c  ",s.c);
       printf("\nA = %d  ",s.a);
       s.a = 65;
       printf("\nNew C = %c  ",s.c);
       printf("\nNew A = %d  ",s.a);
      getch(); }
```

C = z

A = 122   /*This is not same for all execution*/

New C = A

New A = 65

First time the union variable **a** has some unexpected data. After changing its value the character variable **c** value also has been changed as from **'z'** to **'A'**. This is enough to prove whether the memory in the union is shared or not.

Typedef inition **:**

This is also a user defined data type used to set a new name for the existing data types. Are you feeling in the understanding of the word **int** instead of integer. If so, leave worries. The **typedef** statement is used to create a new user defined data type. That is we can give a new name for the data types like int, float etc. The declaration is similar to the simple variable declaration. The general format of the declaration is

```
typedef   data-type   new-name;
```

In feature to declare the same kind of data type we can use the **new-name** instead of old **data-type**. Look the following example:

        typedef int number;

Here **number** is declared as an **int**eger data type and it is equivalent to the data type **int**. Now we can use **number** to declare variable of integer type.

        number a,b,c;

By using the **typedef** the new data type **string** will be created as follows with the example. There   is   no   provision   for   declaring   string   directly.

```
/* Example for typedef declaration */
main( )
{
        typedef  char string[80];
        string  name;
        /* name is string type data  */
        printf("\nEnter a name : ");
        scanf("%s",name);
        printf("\n'%s' welcome to all",name);
}
```

> **Enter a name : Sanjai**
>   'Sanjai' welcome to all

Enumerated data type:

Enumeration is also another type of user-defined data type, for which we are allowed to specify the possible values for the test. We can utilize this feature to keep some names instead of values. In some cases remembering the numeric value is difficult. String or Word is always better instead of the numbers.

For example, in C programming language, the numeric value **0** (Zero) is treated as **FALSE** and **1** is treated as **TRUE**. When we use these values like **0** or **1**, we may confuse little bit. If the number will increases the problem also increase.

Format of the Enumerated definition is

    enum tag

        {

                Constant-Name1=Value1,

                Constant-Name2=Value2 . . .

        } variable(s);

The following is a simple example,

    enum status

     {

       FALSE,TRUE

     };

Here the user defined data type **status** is created and its value may be **FALSE** or **TRUE.** In this case, as I mentioned in the introduction   the value of FALSE is actually 0 and the value of TRUE is 1.

We can change the values by specifying its value explicitly. For example the declaration

    enum status

    {

        TRUE=1,FALSE=2

    };

Here TRUE will be interpreted as value 1 and FALSE as 2. One more example, to keep the days of the week. The days are mentioned like sun,mon,tue ....sat. But there is no constant values like this

for our representation. We have to use some values like 0 to represent sun, 1 to represent mon, 2 represent tue etc.

Another way of keeping the days of the week is as follows using the enumerated declaration.

> enum days
>
> {
>
> SUN, MON, TUE,WED,THU,FRI,SAT
>
> }dow;

Here the variable dow (day of the week) may contain any one of the value given values (SUN,MON ...) and its actual interpretation is 0,1,2 ....   etc.

The following is a simple program to check the value of the constant name in the enumerated data type.

```
/* Example for enumerated type */
enum status
  {    TRUE,FALSE   };
main()
{         enum status value;
printf("%d",TRUE);      }
```

Bit fields :

C permits us to use small bit fields to hold data. We have been using integer field of size 16 bit to store data. The data item requires much less than 16 bits of space, in such case we waste memory space. In this situation we use small bit fields in structures.

The bit fields data type is either int or unsigned int. the maximum value that can store in unsigned int filed is :- (2 power n) – 1 and in int filed is :- 2 power (n – 1) . Here 'n' is the bit length.

Note :

scanf() statement cannot read data into bit fields because scanf() statement, scans on format data into 2 bytes address of the filed. Bit fields do not have addresses—you can't have pointers to them or arrays of them.

Syntax :

    struct struct_name

    {

        unsigned (or) int identifier1 : bit_length;

unsigned (or) int identifier2 : bit_length;

……………………………………….

……………………………………….

unsigned (or) int identifierN : bit_length;

    };

Program : bit_stru.c

```c
#include<stdio.h>
#include<conio.h>
struct emp
{
 unsigned eno:7;
 char ename[20];
 unsigned age:6;
 float sal;
 unsigned ms:1;
};
void main()
{
 struct emp e;
 int n;
 clrscr();
 printf("Enter eno : ");
 scanf("%d",&n);
 e.eno=n;
 printf("Enter ename : ");
 fflush(stdin);
 gets(e.ename);
 printf("Enter age : ");
 scanf("%d",&n);
```

```
e.age=n;
printf("Enter salary : ");
scanf("%f",&e.sal);
printf("Enter Marital Status : ");
scanf("%d",&n);
e.ms=n;
clrscr();
printf("Employ number  : %d",e.eno);
printf("\nEmploy name    : %s",e.ename);
printf("\nEmploy age     : %d",e.age);
printf("\nEmploy salary  : %.2f",e.sal);
printf("\nMarital status : %d",e.ms);
getch();
}
```

POINTERS

Introduction

The word pointer is not a new word for the people and we are using this word in different places with different interpretations. People are always have some wrong opinion about pointers, like it is very tough to understand and hard to use etc. Why? What is in a pointer? Nothing to fear, it has lot of advantages than others. This chapter will relieve you from fear and enjoy with the pointer and its applications.

What is pointer?

- It is a powerful feature of C Language
- It is a new kind of data type
- It stores the addresses, not values
- It allows indirect access of data
- It allows to carry whole array to the function
- It will help in returning more than one value from function
- It helps for dynamic memory allocation

What is pointer in our regular life? It is an indicator, which helps to reach particular place. It is also like a symbol, marker, etc. Look the following and find how the pointer is helping the people to precede towards Coimbatore, using the Hand symbol.

☞Way to Coimbatore
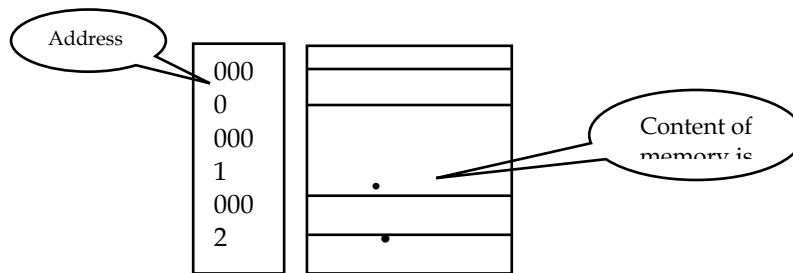
We know some basics regarding the variable declaration and how the memories are being allotted for them. Memory is divided into small pieces to keep small data called byte (8 Bits). Our program and data will be stored in somewhere in the memory where the free area is available. The variables are the names used for reference but everything internally referred by the memory address. Let us see the following diagrams and how the memory is allocated for the variables.

There is a declarative statement   char ch = 'A' ;

At the time of execution the compiler will make following process.

⇨ One byte memory is reserved for the character variable ch

⇨ and store the character value 'A' in that memory location

The memory allocation may be as follows
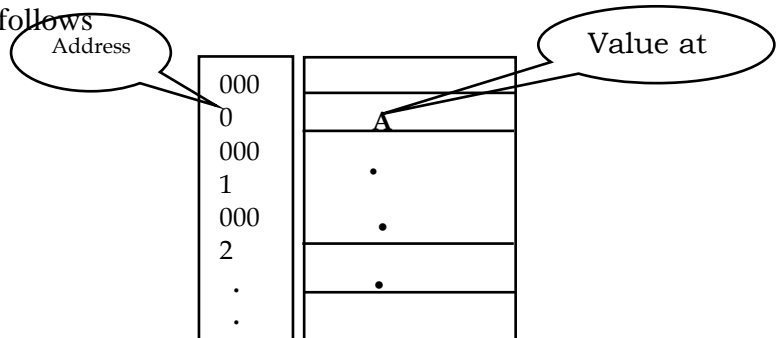
The character variable ch is stored 0001 and the value of that location is 'A'. The address of the variable is not constant and it may vary for next execution.

> The address of **ch** is not a constant for every execution and for any user-defined variable the address is not constant

Operators in pointers:

The pointers will help us in doing variety of operations and applications. The two essential operators are given below.

1. * -> Indirection operator, which used to retrieve the value from the memory location
2. & ->Address operator, which is used to obtain the address of variable

Now the above two operators will help in viewing address and values. Consider the declarative statements

int a =10;

- ⇨ If we refer a, it returns the value of a as 10
- ⇨ If we refer &a, it returns the address of a, that is where the memory is allocated for this variable and
- ⇨ *(&a) refers to the value of a. Because &a refers the address of a and *(&a) means that value at address of a

If we test the previous idea via a program, you may be happier. Execute the following program and realize about the address is retrieval.

```
/* Program to collect the address of variable */
main( )
{  int a=10;
  printf("\n Value of a = %d ",a);
  printf("\n Memory address of  a = %u",&a);
}
```
Value of a = 10

Memory address of a = 8716

Note:

> Memory address is always a positive value. So we can use format string character **%u** for printing the address.

How to declare the pointer variable?

No need to worry about the declaration of pointer variable, it can be declared as simple variable declaration with small change as follows.

```
Data-type  *pointer-variable;
```

Here, the character '*' indicates that the variable is pointer variable. For example, the declarative statement:

int *ptr;

Here ptr is a pointer variable and It will point to one integer memory location.

Before any operations on pointer variable, we must store the address, because the pointer variables will have address not values.

int *ptr;

ptr = 10;

The compiler will show an error, because we can't store value in a variable in this manner. We can assign the direct address or address of the variable to the pointer variable as follows.

int   a=10;

int *ptr;

ptr = &a;   /* Address of a is assigned to ptr */

Now the address of a is assigned to the pointer variable ptr. So, ptr will point the same memory location where a points to.

ptr = 0x41700000;   /* Direct Address, Hexadecimal */

This assignment statement is direct address assignment and 0x41700000 is an address not a value. So we can assign the address to the pointer variables in any one of the manners.

How to retrieve the values from the memory? We know that the * operator will help here. Yes. If we know the address of a, then without the assistance of variable a we can access the values of a and its illustration is as below.

int a=10;

int *ptr;

The following is a pictorial representation of the above declarative statements. Before processing the assignment statement the memory representation is as follows.
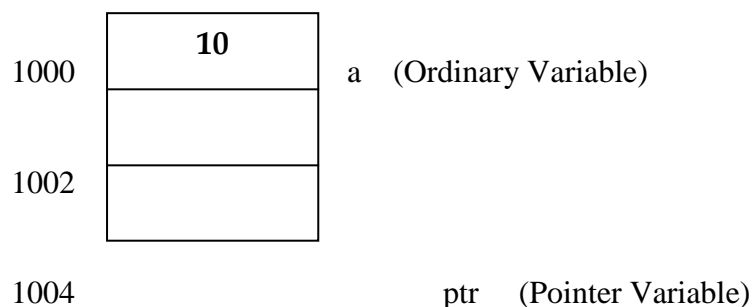
From this diagram we can conclude the following

⇨ Address of a is 1000.

⇨ Value at memory location 1000 is 10.

⇨ Address of pointer variable ptr is 1004.

After the assignment statement   ptr = &a, the diagram is as follows

1000	10	a (Ordinary Variable)
1002		
	1000	
1004		ptr (Pointer Variable)
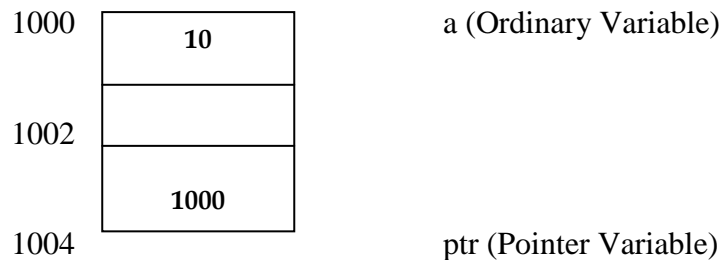
From this diagram we can get the following information

⇨ Address of a is 1000 and value at memory location 1000 is 10.

⇨ ptr holds the address of   a (i.e. 1000).

⇨ So, ptr points to a indirectly and

⇨ memory address of ptr is 1004

If we refer the value stored at ptr by *ptr, we may expect the result as 1000. But 1000 is not a value and it is an address of variable a. So, *ptr returns the value stored at location 1000, and returns the value 10, which is a value of a. The following program is illustrating the previous theoretical discussions.

```
/* Accessing values indirectly using pointers */
main( )
{ int  a=10;
 int *ptr;
ptr = &a;
/*Address of a is assigned to pointer variable ptr*/
printf("\n Value of a = %d ",a);
printf("\n Value of a = %d ",*ptr);
printf("\n\nMemory address of variable a   = %d ",&a);
printf("\nMemory address of variable a   = %d ",ptr);
printf("\nMemory address of variable ptr = %d ",&ptr);
}
```

Value of a = 10

Value of a = 10

Memory address of variable a = 1000

Memory address of variable a = 1000

   Memory address of variable ptr = 5000

From the above program we can come to the conclusions that the value of a can be accessed by referring a and using the pointer variable by *ptr.

Operations on pointer – Indirect Modification

   What we have discussed so far is about the fundamental idea of pointers. Now we are clear about how to use pointer variable and access the value of any variable indirectly. Pointer purpose not only stops with these operations and also is able to change the value of the specified memory locations indirectly.

int a=10;

   int *ptr=&a; /* Address of 'a' is assigned to 'ptr' */

ptr =100; / Value of 'a' is changed indirectly */

We are able to refer the value of any variable indirectly without the help of that variable. The changes on a variable can also be made without using that variable. The following program illustrates the indirect change of value of variable.

```
/* Program for changing values indirectly */
main( )
{    int  a=10;
   int *ptr;
   ptr=&a;
   printf("\nOld Value of a = %d ",a);
   *ptr=100;
   printf("\nNew Value of a = %d ",a);
}
```
 Old Value of a = 10
 New Value of a = 100

In the above program we have not made any change in the value of a directly. But the statement *ptr=100 changes the value of a as 100. Because the variable ptr is pointing to the memory address of a. So, we changed value of a indirectly.

Pointers and Expressions:

With the help of simple arithmetic operations a pointer variable can travel any location in the memory and consider the following as a memory structure for our discussion.

 1000   1001   1002   1003   1004   1005   1006   1007

The declaration

 int *ptr ;

Assume that the starting address of integer pointer variable ptr is pointing to the first memory address 1000. If we increment the pointer variable ptr by 1, we may expect ptr will becomes 1001. But it is not correct? Oh! Why? The variable ptr is an integer pointer variable. Each integer requires two memory locations. So every increment in ptr will point to next integer memory location, here it is 1002. Suppose ptr is a character pointer variable, for every increment of ptr, it will be pointing to the adjacent memory location, because char need 1 byte memory. Look the following examples.

int *ptr;

 ⇨ Assume ptr is now pointed the location 1000.

ptr++;

 ⇨ After this statement ptr is pointed to the location 1002

ptr--;

 ⇨ Now ptr is adjusted to the previous location 1000.

ptr = ptr+3;

 ⇨ ptr is now at the location 1006

Note:
> ⇨ The pointer variables are always adjusted to the next memory location depending on its data type.
> ⇨ Operations other than addition and subtraction are not possible

Pointers and Arrays - Single Dimensional

Array is a collection of same elements and is stored in continuous memory locations. Are you able to prove the last point, stored in continuous memory locations? You can prove this statement when you execute the following program. Assume that the following array elements are stored in memory as below

 int a[5] = {10,20,30,40,50 };
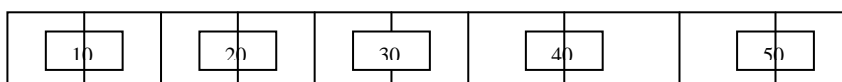
10	20	30	40	50

1000   1001   1002   1003   1004   1005   1006   1007   1008   1009   1010

/* Program to check the definition of array   */

  main( )

  {   int i, a[5] ={10,20,30,40,50};

    for(i=0;i<5;i++)

printf("\n%d is stored at location %d ",a[i],&a[i]);   }

---

10 is stored at location 1000

20 is stored at location 1002

30 is stored at location 1004

40 is stored at location 1006

50 is stored at location 1008

What is base address of array? How to obtain the same? Consider the following declaration and see how the base address or starting address of the array will be obtained.

    int a[5] = {10,20,30,40,50};

First element of array is referred by a[0] and its address is by &a[0]. Here &a[0] refers to the starting address or base address of the array a. Otherwise the name of the array itself refers the base address, i.e. a. Once we know the starting address of array, we can travel through all the elements of the array easily by making simple arithmetic operation.

int *ptr;

        ptr = &a[0];   /* &a[0] refers to the starting address of array */

                        (or)

ptr = a;   /* a also refers to the starting address */

The first element is referred by *ptr   (or)   *(ptr+0).

Second element is referred by   *(ptr+1).

Third element is referred by   *(ptr+2).

Fourth element is referred by   *(ptr+3) and in common, element is referred by *(ptr+i).

The following program is an example for processing the array elements using the pointer variable.

/* Program to process the array using pointers */

main( )

{

```
int  a[5] ={10,20,30,40,50};
int i, *ptr;
/*Starting address of array is assigned*/
ptr=&a[0];
    for(i=0;i<5;i++)
     printf("\n%d is stored at location:%d",
                    *(ptr+i),(ptr+i));
   }
```

10 is stored at location : 1000

20 is stored at location : 1002

30 is stored at location : 1004

40 is stored at location : 1006

50 is stored at location : 1008

Now we are going to sort the numbers using pointers. We are also finding the maximum and minimum from the set of numbers after sorting.

```
        /* Program to sort numbers using pointers  */
        main( )
{
 int  a[15],n,i,j,temp,*ptr;
 printf("\nHow many numbers ");
 scanf("%d",&n);
 printf("\nEnter %d values\n",n);
 for(i=0;i<n;i++)
                scanf("%d",&a[i]);
  /* Starting address of a is assigned to ptr*/
 ptr = a;
 printf("\nValues before sorting\n");
 for(i=0;i<n;i++)
                printf("%d\t",*(ptr+i));
     for( i = 0 ; i<n-1 ;i++)
         for(j =i+1 ; j<n; j++)
           if  (*(ptr+i) > *(ptr+j))
             {
```

```
          temp  = *(ptr+i);
                  *(ptr+i) = *(ptr+j);  /* Swaping    */
          *(ptr+j) =  temp;
              }
 printf("\nValues after sorting\n");
 for(i=0; i<n; i++)
          printf("%d\t", * (ptr + i));      }
```

How many numbers 5

Enter 5 values

22 55 11 44 33

Values before sorting

22 55 11 44 33

Values after sorting

11 22 33 44 55

Pointers and Strings:

String is a collection of characters and it can be also called as character array. In the previous topic we discussed many programs using numbers. The pointers are beneficial in character-based application also. The character pointer variable is declared as follows

char *ptr;

Here *ptr is a pointer variable, which points to the array of characters. The string value can be assigned to the variable as below

        char name[ ]="Karthikeyan";

The starting address (base address) of the string is taken any one of the following ways with the help of above declaration

    name (or) &name[0]

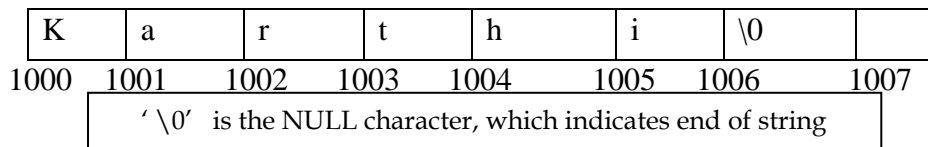   /* Both are points to the starting address of the string */

The statements

                        char *ptr;

char   name[ ] ="Karthi";

 are declarative statements and the assignment statement is

ptr = name;

Here the starting address of the character array variable or string variable name is assigned to the pointer variable ptr. Now both name and ptr points to the same memory location. The following diagram illustrates the above.

K	a	r	t	h	i	\0	
1000	1001	1002	1003	1004	1005	1006	1007

' \0' is the NULL character, which indicates end of string

Now we are going to see how to access the characters of the string variable using pointers.

```
/* Accessing string values using pointers */
main( )
{       char  *ptr, name[]="Karthi";
     int  i , l;
         ptr = name;
        /*Starting address is assigned to ptr*/
 l=strlen(name);


         for(i=0;i<l;i++)
            printf("%c",*(ptr+i));
     }
Karthi
```

How to use pointer variable to read a string value? Test the following simple program.

```
#include <stdio.h>
main()
{
char *s;
printf("\nEnter a string : ");
gets(s);
printf("\nYour given string is : ");
while(*s)
   printf("%c",*s++);
getch();
 }
Enter a string : You are welcome
```

Your given string is : You are welcome

The program given below implements the strcpy( ) function, it is used to copy the content of one string to another sting variable.

```
    /* Implementation of strcpy command */
main( )
{       char s1[15],s2[15],*ptr;
        int i,j,l;
        printf("\nEnter a source string  : ");
    gets(s1);
    ptr = s1;
    l=strlen(s1);
    for(i=0;i<l;i++)
            s2[i] = *(ptr+i);
    s2[i]='\0';
    printf("\nCopied string  :%s",s2);
  }
```

Enter a source string : karthi

Copied string :karthi

Pointers and Functions:

We discussed the importance of function in a program and how the same is used in various applications in the previous chapter. The drawback of simple function is that, we can't return more than one value from it. The change made in the called function does not reflect in the calling function. (Calling function – A function from which the new function is invoked and the Called function – A function to which the control has to be transferred). One more problem is that we can't pass the entire array to the function. The problem of function is explained by using the following program.

```
/* Testing the values of changes */
main( )
{       int a=10;
        printf("\nBefore change : %d ",a);
        change(a);
        printf("\nAfter   change : %d ",a);
```

```
        getch( );
}
void change( int b )
{       b=100;
        printf("\nInside the function : %d ",b);}
```

Before change : 10

Inside the function : 100

After   change : 10

In this program the value of a in main() is 10 and it is passed to the user defined function change( ). The function receives the value of a via b. Inside the function the value of b has been changed as 100. But this change will affect only in b not in the value of a, because b is local to the function. The value of a has been copied to b. This is equivalent to the statement b=a; So any changes in b will never affect the value of a here. Go ahead and read the next topic to solve these problems.

How to change the value using function?

Are you able to change the value of argument in the called function? If so, how? Using pointers you can achieve this. You can pass the address of a variable to the calling function and so the changes made in the called function will be reflected in the calling function. The following program example illustrates this idea.

```
/* To changes the values of variables using pointers */
main( )
{int a=10;
printf("\nValue before change = %d ",a);
change(&a);   /* Passing address of a*/
printf("\nValue After  change = %d ",a);
}
void change (int *b)
{              *b=100;    /* Changing values indirectly */
}
```

Value before change = 10

Value After change = 100

How the value of a have been changed here? From the main() we are passing the address of a to the function by the statement

change(&a); /* Address of a   is passing */

Now we are passing the address of a, not the value of a. So the address must be received by the pointer variable only and the function definition will be

void change(int *b)

At the time of execution the address of a has been assigned to the pointer variable b, which is equivalent to the following statement

int *b;

b = &a;

Now both a and b is pointing to the same memory location and any change made in b will automatically affect the value of a.

Same variable name in many part of the program. Confusion. The name of the variable in one function may be same in another function. The variable name is only for the user reference not for the system. This problem is clearly presented in the following program.

```
/* Getting address of variable */
main( )
{  int a=10;
 printf("\nAddress of 'a' in  main  : %d ",&a);
  change( ) ;
}
void  change( )
{        int a=100;
        printf("\nAddress of 'a' in function: %d ",&a);}
```
```
Address of 'a' in  main  : 5000
Address of 'a' in  function : 7000
```

In this program there are two variables with same name as a. One is in main( ) and another is in the user defined function change( ). For every declaration the memory allocation for each variable is different from others. So that the result of the above program is 5000 and 7000, two different addresses even though names are same.

Call by Value & Call by reference

A function can be invoked by so many ways as we discussed in the previous chapters. The way of calling function can be classified into two,

1. Call by value

2. Call by reference

Here is a program, which finds the sum of two numbers illustrates the above ways.

Call by value:

This can be done in two ways either using a variable or directly passing a value.

```
/* Passing values to the function */
main( )
{int a=10,b=20,c;
c=sum(a,b); /* Passing the value  of a,b */
printf("\nSum  = %d ",c);}
int sum( int  x,  int  y)
{       return ( x + y);}
```

In this program the value of a and b has been passed to the function to find the sum. It's just like the following simple assignment statement

x = a   and   y = b;

We can pass the value to the function by giving direct value also.

sum(10, b);   sum (10,20)

Call by reference:

What is reference? In some occasions, people may want to clarify about others using the reference in the real life. Here the variables are indirectly using the reference instead of direct involvement. So the function can also be invoked by using the reference that is addresses (Using pointers). The previous program with simple modification using reference.

```
 /* Example for Call by Reference */
main( )
{
int a=10,b=20,c;
c = sum( &a, &b);  /* Passing address of a, b */
printf("\nSum  = %d ",c);
}
 int sum (int   *x,  int *y)
 {
        return (*x+*y);
 }
```

What is the difference between the previous two programs? In the first one values are passed to the function in a simple manner. But in the second one, address (i.e. reference) of those variables is passed.

Passing array to the function:

In general we are not allowed to carry the whole array to the function. We can pass the elements one by one. If we need to process the whole array at the same time, this provision will not help. Now the hidden features of pointer will be used to carry the entire array without much more risks.

How it is possible? It is very simple. Array elements are always in the continuous memory locations. First you obtain the base address of the array. If we get the starting address of the array, we can reach any element in the array by making simple arithmetic operations. For the function side, just we have to pass the base address of array to the function. This idea is illustrated in the following program.

```
/* Passing array to the function using pointers  */
main( )
{  int a[ ]={10,20,30,40,50};
   display(a);
   /* Passing the base address of array */
}
display (int  *x)
{    int i;
   printf("\n Array elements are   : ");
   for(i=0;i<5;i++)
        printf("%5d",*(x+i));
}
Array elements are :   10  20  30  40  50
```

In the above program the starting address of array is passed to the function by the statement

display(a);

The function will receive the starting address of array by defining the function argument as follows.

display(int *x)

This is equivalent to the assignment as x=a; The following is a program to test the previous idea by sending an array elements to the function and the elements are doubled in the function. Finally the changes are ensured by displaying the values in main() function.

```
/* Program to pass the whole array to the function */
#include <stdio.h>
main()
{int i, a[5]={10,20,30,40,50};
clrscr();
printf("\nElements before invoking function : ");
for(i=0;i<5;i++)
        printf("%5d",a[i]);
test(a);
printf("\nElements after invoking function : ");
for(i=0;i<5;i++)
        printf("%5d",a[i]);
getch();
}
test(int *x)
{ int i;
 for(i=0;i<5;i++)
  *(x+i) = *(x+i) * *(x+i);
}
```

Elements before invoking function : 10 20 30 40 50

Elements before invoking function : 100 400 900 1600 2500

I hope now you have an idea about how the array elements are carried to the function. Here is a program to find the mean, variance and standard deviation of N floating point numbers. Formula to calculate the standard deviation is

$$\text{Standard deviation} = \sqrt{\text{variance}}$$

Where

$$\text{Variance} = 1/n \sum (X_i - \text{Mean})^2 \quad \text{and} \quad i = 1 \text{ to } n$$

$$\text{Mean} = 1/n \sum X_i$$

So , to find standard deviation the following is the general steps.

1. Find the sum and mean
2. Find the variance   and
3. Finally calculate the standard deviation.

/*Program to find the standard deviation using pointers */

```
#include <math.h>

main( )
{    float a [ ]={1.1,2.2,3.3,4.4,5.5};

     sd(a);

}

void sd (float *x)

{

int i;

float s1=0,s2=0,s3=0,sddev,mean,var,temp;

printf("\nValues : ");

for(i=0;i<5;i++)

{

        s1 += (*(x+i));   /* Finding  summation */

        printf("%5.2f\t",*(x+i));

}

mean = s1/5;    /* Calculation of   Mean   */

for (i=0;i<5;i++)

{       temp=*(x+i)-mean;

        s2+=pow(temp,2);

}

var   = s2/5; /* Calculation of variance */

sddev = sqrt(var);  /* Calculation of  S D */

printf("\nMean         =   %5.2f",mean);

printf("\nVariance     =   %5.2f",var);

printf("\nStd.Deviation  =   %5.2f",sddev); }
```

```
Values : 1.10  2.20   3.30   4.40   5.50

Mean        =   3.30

Variance    =   2.42

Std.Deviation  =   1.56
```

2 D Array & Pointers

As mentioned about the pointer, it gives a very good support to arrays including two-dimensional array. Matrix is a traditional and very famous example for a two dimensional array. Consider the following declarative statement

int   a[2] [3] ;

This declaration tells

⇨ a   is a two dimensional array

⇨ Maximum number of elements are 6 (2 x 3 = 6)

⇨ and all are integer type of data .

⇨ So, each element occupies 2 bytes (Totally 12 bytes)

The values for the above two-dimensional array are initialized as below and its corresponding memory allocation is illustrated.

int a[2][3] = {

        {10,20,30},   First row

          {40,50,60},   Second row

      };

10	20	30
1000	1002	1004

40	50	60
1006	1008	1010

Two-dimensional array is a collection of single dimensional arrays and each single array is pointed by the pointer variable. Here a[0] points to the first single dimensional array, a[1] points to the second single dimensional array etc.

a [0]   can be rewritten as *(a+0) and

a [1]   can be rewritten as *(a+1) etc.

So, a[0] refers to the starting address of first array, and its values are referred as,

a[0][0]   =>   First row first column

a[0][1]   =>   First row second column etc.

a[0][0] can be referred as *(*(a+0)+0))

 *(a + 0) => Starting address of first array i.e. a[0]

(*(a+0)+0)   => address of first row's first element i.e. &a[0][0]

((a+0)+0) => Value of first row's first element   i.e. a[0][0] (* is the value at the location operator)

Value returned by a[0] is   1000.

Value returned by a[1] is   1006.

Value returned by &a[0][0] is   1000

Value returned by &a[0][1] is   1002   etc

The following is an example program to check the starting address of each array.

```
/* To get the base addresses of 2D Array  */
main( )
{    int a[2][3]={ {10,20,30},  {40,50,60},   };
    int i;
 for(i=0;i<2;i++)
  printf("\nBase address of %d array :%u",i+1,a[i]);
}
```

Base address of 1 array : 1000

Base address of 2 array : 1006

One more program is here to give more idea about two-dimensional array and pointers.

```
/* Accessing the elements of array */
main( )
{       int a[2][3]={ {10,20,30}, {40,50,60}, };
int  i , j;
        for(i=0;i<2;i++)
                for(j=0;j<3;j++)
printf("\na[%d][%d] = %d  is stored at:
     %u", i,j, a[i][j], &a[i][j]);
     getch( );}
```

a[0][0] = 10 is stored at : 1000

a[0][1] = 20 is stored at : 1002

a[0][2] = 30 is stored at : 1004

a[1][0] = 40 is stored at : 1006

a[1][1] = 50 is stored at : 1008

a[1][2] = 60 is stored at : 1010

Referring the elements of two-dimensional array is little bit difficult than a single dimensional array. The following is an example for this reference, which is based on the previous declaration and its addresses. We are going to refer the element at a[1][2], the pointer notation is as follows. Here i is 1 and j is 2.

((a + 1)+ 2)

    1. *(a+1)

        ⇨ It returns the starting address of second array equal to a[1] and it returns 1006

    2. (*(a+1)+2)

        ⇨ The value returned by *(a+1) will be incremented by 2. So it returns the address 1010.

    3. *(*(a+1)+2)

        ⇨ It returns the value of that location. i.e. Value at location 1010 is 60.

    The next program illustrates how to access the elements of a two-dimensional array using pointers

```
/* Accessing the elements of 2D using pointers */
main( )
{
int a[2][3]={
        {10,20,30},
        {40,50,60},
        };
int  i , j;
for(i=0;i<2;i++)
  for(j=0;j<3;j++)
    printf("\n%d is stored at : %u",
            *(*(a+i)+j),(*(a+i)+j));
        getch( );
    }
```

10 is stored at : 1245032

20 is stored at : 1245036

30 is stored at : 1245040

40 is stored at : 1245044

50 is stored at : 1245048

60 is stored at : 1245052

Array of pointers

What is the use of array? Array is used to store number of elements in a single variable. The elements may be of any type. But all of them must be of the same type. We have discussed many programs using arrays with different type of values like integer, real and character etc.

Can we store addresses as array elements? Yes. We can. Instead of simple data, the address can be stored. The way of declaring array of pointer is explained in the following.

        int *ptr;

Here ptr is a pointer variable, which points to one integer memory location. With small change in the above declaration, the statement is

        int *ptr[5];

   Here ptr is variable and it is allowed to have addresses of 5 variables not values. In this case we can store 5 different integer addresses to this array variable. Elements of the array may contain different addresses.

 int a,b,c;

        ptr[0] = &a;

   /* Address of a is assigned to first element of array */

     ptr[1] = &b;

   /* Address of b is assigned to second element of array */

  ptr[2] = &c;

The value of a can be referred as *ptr[0]. The following is a program gives an idea of our discussion.

```
      /* Example for array of pointers */
    main( )
     {
int a=10,b=20,c=30;
int *ptr[5];        /* Array of  pointers */
clrscr( );
ptr[0]=&a;
ptr[1]=&b;
ptr[2]=&c;
      /* Value of pointer variable is accessed  */
 printf("\na = %d ",*ptr[0]);
 printf("\nb = %d ",*ptr[1]);
 printf("\nc = %d ",*ptr[2]);
 printf("\n Address of a    = %u",&a );
```

```
printf("\n Value  of ptr[0]= %u",ptr[0] );
    }
```

```
        A=10
        B=20
        C=30
Address of a      =   12042
Value  of ptr[ 0 ] =   12042
```

In the above program ptr[0] holds the address of variable a. So &a and ptr[0] contains the same values (ie address).

Calling functions using Pointers:

Normally functions are invoked by specifying its name with necessary arguments. Now we are going to invoke the function using pointers. The address of variable can be obtained as follows.

```
  int a;
  printf("\nAddress = %u ",&a);
```

The output of above would be address of the variable a. It may be 1240, which is not always same. Address of the function can also obtained as illustrated below.

```
/* Obtaining the address of function */
main( )
{  int test( )
 printf("\nAddress  function  test = %u ",test);
 }
```

This program returns the address of function test(). This address can be assigned to a pointer of the function variable as like below.

int test( );   /* Function prototype declaration */

int (*ptr)( ); /* Pointer to function */

Address of function test() is assigned to the pointer variable ptr as

 ptr = test;

The function can be invoked using pointer as below

```
            (*ptr)( ); /* Similar to calling as test( )  */
```

The following a complete program, which illustrate the above discussion like how the functions are called using the pointers.

```
/* Illustrating function calling using pointers  */
```

```
main( )
{            void test( );
       void (*ptr)( );
              ptr = test;      /* Address  assignment  */
       (*ptr)( );       /* Function   Calling      */
}
void  test( )
{       printf("\nHello ");}
```

Returning address

The previous section provides an idea about the function and pointers that indirectly returns the address. But we can return a memory address to the calling function as like a normal function return type. Look the following code

     int Read()   => The function returns an integer value

     char Read()   => The function returns an character value

     void Read()   => The function returns nothing

     int * Read()   => Now the function returns memory address.

The following example program illustrates the idea of returning an address from the function. The program read the array elements in the function and returns the base address of the array to the main() function. Later the address will be used in further process in main() function.

```
     /* Program  which  read  value  in  function  Read()  and  return  the
address to the main() function  */
#include <stdio.h>
#include <conio.h>
int * Read(int);
main()
{
int *a,n,i;
clrscr();
printf("\nEnter the size of the array :");
scanf("%d",&n);
a = Read(n);
```

```
printf("\nArray elements are \n");
for(i=0;i<n;i++)
 printf("%5d",*(a+i));
 getch();
}
int * Read(int m)
{
int *p,i;
p=(int *) malloc(sizeof(int) * m);
printf("\nEnter %d values ",m);
for(i=0;i<m;i++)
  scanf("%d",(p+i));
return p;}
```

Structures and Pointers:

The features of pointers are not limited with simple application. It is used in the structure also. The declara[...]

```
struct  structure-tag  * structure-pointer;
```

Proceed with the following example and see how an ordinary variable and pointer variables are used in the program.

```
struct stu *s1;  /* s1 is structure pointer */
struct
 {
      char name[15];
      int  rollno;
 }s1, *s2;
```

In the above declaration s1 is an ordinary structure variable, but s2 is a pointer to structure variable. The members of the structure s1 will be referred using dot (.) operator. But the members of pointer to structure variable will be accessed using an operator called an arrow operator (→). In simple definition, instead of dot(.) operator we have to use the arrow operator. So the members of s2 are referred as s2→name and s2→rollno. The following program illustrates our discussion and may be clarified easily.

```
/* Example for pointers and structures */

main( )
{  struct stu
  {
    char name[25];
    int rollno;
};
struct  stu s1, *s2;
printf("\nName of the student  : ");
scanf("%s",s2->name);
printf("\nRoll No.          : ");
scanf("%d",&s2->rollno);
    printf("\nName: %s\nRoll No :%d ",s2->name,s2->rollno);
  }
```

POSSIBLE QUESTIONS – UNIT III
Part-A

Online Examinations   **(One marks)**

1. _____is a collection of objects of similar type

     a) Objects   b) methods   **c) classes**   d) messages

2. The _____is an entry-controlled loop

     a) while   b) do-while   c) for   d) switch

3. Which of the following function / type of function cannot be overloaded?

     a) Member function   b) Static function

     c) **Virtual function**   d) Both B and C

4. Which is used to keep the call by reference value as intact?

     a) static   **b) const**   c) absolute   d) none

5. By default how the value are passed in c++?

     a) call by value   b) call by reference   c) call by pointer   d) none

6. Which of the following is a two-dimensional array?

     a) array anarray[20][20];   **b)** int anarray[20][20];

     c) **int array[20, 20];**   **d)char array[20];**

7. Which reference modifier is used to define reference variable?

     a) &   b) $ c) #   d) none

8. Which of the following are themselves a collection of different data types?

     a) String   b)**Structure**   c) Char   d) All

9. Union differs from structure in the following way

     a) All members are used at a time

     b) Only one member can be used at a time

     c) Union cannot have more members

     d) Union initialized all members as structure

10. The _____ functions are used to handle the single character I/O operation.

     a) get() and put()   b)clrscr() and getch()   c) cin and cout   d) None

Part-B   2 MARKS

1. What are the features and uses of pointers?

2. Define structure.

3. Define union.

4. Write about array of structure.

5. Define pointers with example.

6. What are pointers to pointers?

7. How will you declare pointer variable.

8. Pointers Vs References.

9. What is derived data type?

10. What is the purpose of & and * operators?

11. Discuss about pointers and arrays with example.

12. Distinguish between (*m) [5] and *m[5].

Part-C   6MARKS

1. Explain pointers to pointers with suitable program.

2. How to declare and initialize structure and unions?

3. Discuss pointers to structures with example.

4. How to pass and return structures from functions?

5. Mention the difference between pointers vs. references

6. Write note on passing pointers as function arguments.

7. With proper example explain Array of structures

8. With proper example explain pointers in c++.

9. Explain Unions with simple example.

10. Write a program to swap two numbers using pointers.

11. Enumerate in detail about the array of structure with example.

12. Discuss the methods used to send pointers to functions with an example.

13. How will you declare, initialize and access a union? Explain in detail with example.

Part -A Online Examinations   **(1 mark questions)**

SUBJECT: PROGRAMMING   SUBJECT CODE: 17CSU101
FUNDAMENTALS USING C/C++

UNIT-III

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	calling function	declaration	objects	none of the above	declaration
4	The class _____ describes how the class function are implemented	Function definition	declaration	arguments	none of the above	Function definition
5	The keywords private and public are known as _____ labels	Static	dynamic	visibility	const	visibility
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 declared inside the class are called as _____	Function variables	data members	member function	data variables	data members
9	The symbol _____ is called the scope resolution operator	>>	::	<<	::*	::
10	A member function can call another member function directly without using the _____ operator	Assignment	equal	dot	greater than	dot
11	A _____ member variable is initialized to zero when the first object of its class is created	Dynamic	constant	static	protected	static
12	_____ Variables are normally used to maintain values common to the entire class.	Private	protected	Public	static	static
13	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
14	The _____ member variables must be defined outside the class.	Static	private	public	protected	Static
15	A friend function, although not a member function, has full access right to the _____ members of the	Static	private	public	protected	private
16	_____ enables an object to initialize itself when it is created	Destructor	constructor	overloading	none of the above	constructor
17	_____ destroys the objects when they are no longer required	Destructor	constructor	overloading	none of the above	Destructor

18	The _____ is special because its name is the same as the class name.	Destructor	static	constructor	none of the above	constructor
19	A constructor that accepts no parameters is called the _____ constructor	Copy	default	multiple	none of the above	default
20	Constructors are invoked automatically when the _____ are created	Datas	classes	objects	none of the above	objects
21	Constructors cannot be _____	Inherited	destroyed	both a & b	none of the above	Inherited
22	Constructors cannot be _____	Destroyed	virtual	both a & b	none of the above	virtual
23	Constructors make _____ calls to the operators new and delete when memory allocation is required	Explicit	implicit	function	none of the above	implicit
24	The constructors that can take arguments are called _____ constructors	Copy	multiple	parameterized	none of the above	parameterized
25	The constructor function can also be defined as _____ function	Friend	inline	default	none of the above	inline
26	When a constructor can accept a reference to its own class as a parameter, in such cases it is	Multiple	copy	default	none of the above	copy
27	When more than one constructor function is defined in a class, then the constructor is said to be	Multiple	copy	default	overloaded	overloaded
28	C++ complier has a _____ constructor, which creates objects, even though it was not defined in the class.	Explicit	default	implicit	none of the above	implicit

					parameterized	
29	A _____ constructor is used to declare and initialize an object from another object	Default	copy	multiple	parameterized	copy
30	The process of initializing through a copy constructor is known as _____ initialization	Overloaded	multiple	copy	none of the above	copy
31	A _____ constructor takes a reference to an object of the same class as itself as an argument	Delete	new	copy	none of the above	copy
32	Allocation of memory to objects at the time of their construction is known as _____ construction	Static	copy	dynamic	none of the above	dynamic
33	We can create and use constant objects using _____ keyword before object declaration.	Static	new	const	none of the above	const
34	A destructor is preceded by _____ symbol	Dot	asterisk	colon	tilde	tilde
35	_____ is used to allocate memory in the constructor	Delete	binding	free	new	new
36	_____ is used to free the memory	new	delete	clrscr()	none of the above	delete
37	Which is a valid method for accessing the first element of the array item?	item(1)	item[1]	item[0]	item(0)	item[0]
38	Which of the following statements is valid array declaration?	int number (5);	float avg[5];	double [5] marks;	counter int[5];	float avg[5];
39	An object is an _____ unit	group	individual	both a&b	none of the above	individual

40	Public keyword is terminated by a _____	Semicolon	comma	dot	colon	colon
41	Private keyword is terminated by a _____	semicolon	comma	dot	colon	colon
42	The memory for static data is allocated only _____	twice	thrice	once	none of the above	once
43	Static member functions can be invoked using _____ name	class	object	data	function	class
44	The _____ doesn't have any argument	constructor	copy constructor	destructor	none of the above	destructor
45	The _____ also allocates required memory .	constructor	destructor	both a & b	none of the above	constructor
46	Any constructor or destructor created by the complier will be _____	private	public	protected	none of the above	public
47	_____ releases memory space occupied by the objects	constructor	destructor	both a & b	none of the above	destructor
48	Constructors and destructors are automatically inkoved by _____	operating system	main()	complier	object	complier
49	Constructors is executed when _____	object is destroyed	object is declared	both a & b	none of the above	object is declared
50	The destructor is executed when _____	object goes out of scope	when object is not used	when object contains	none of the above	object goes out of scope

51	The members of a class are by default _____	protected	private	public	none of the above	private
52	The _____ is executed at the end of the function when objects are of no used or goes out of scope	destructor	constructor	inheritance	none of the above	destructor

KARPAGAM ACADEMY OF HIGHER EDUCATION

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

Coimbatore - 641021.

(For the candidates admitted from 2017 onwards)

DEPARTMENT OF COMPUTER SCIENCE,CA & IT

SUBJECT   : **PROGRAMMING FUNDAMENTALS USING C/C++**

SEMESTER   : **I**   **L T P C**

SUBJECT CODE: **17CSU101**   **CLASS**   : **I B.Sc.CS**   **4 0 0 4**

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.

Suggested Readings:

1. Herbtz Schildt, 2003,*C++: The Complete Reference*, 4th Edition, McGraw Hill.

2. Bjarne Stroustrup, 2013, *The C++ Programming Language,* 4th Edition, Addison-Wesley.

3. Bjarne Stroustroup, 2014, *Programming - Principles and Practice using C++,* 2nd Edition, Addison-Wesley.

4. E Balaguruswamy, 2008,*Object Oriented Programming with C++,* 2nd Edition ,Tata McGraw-Hill Education.

5. Paul Deitel, Harvey Deitel, 2011,*C++ How to Program*, 8th Edition, Prentice Hall.

6. John R. Hubbard, 2000, *Programming with C++,* 2nd Edition ,Schaum's Series.

7. Stefan Bjornander, 2016,*C++ Windows Programming,* Published by Packt Publishing Ltd.

8. Scott Meyers, 2005, *Effective C++,* 3rd Edition, Published by Addison-Wesley.

9. Harry, H. Chaudhary, 2014 ,*Head First C++ Programming: The Definitive Beginner's Guide*, First Create space Inc, 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*, 5th Edition, Published by Addison-Wesley.

12. Debasish Jana , 2014,*C++ And Object-Oriented Programming Paradigm*,Published by PHI Learning Pvt. Ltd.

13. Richard L. Stegman, 2016, *Focus on Object-oriented Programming With C++,*6th Edition ,CreateSpace Independent Publishing Platform,.

14. Andrew Koeni, Barbara, E. Moo,2000,*Accelerated using C++,* Published by Addison-Wesley .

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. http://*www.cplusplus.com/*

6. http://*www.cppreference.com/*

Unit 4

Dynamic Memory Allocation

Instead of define an int variable (int number), and assign the address of the variable to the int pointer (int *pNumber = &number), the storage can be dynamically allocated at runtime, via a new operator. In C++, whenever you allocate a piece of memory dynamically via new, you need to use delete to remove the storage (i.e., to return the storage to the heap).

The new operation returns a pointer to the memory allocated. The delete operator takes a pointer (pointing to the memory allocated via new) as its sole argument.

For example,

// Static allocation

int number = 88;

int * p1 = &number; // Assign a "valid" address into pointer

// Dynamic Allocation

int * p2;// Not initialize, points to somewhere which is invalid

cout<< p2 <<endl; // Print address before allocation

p2 = new int;   // Dynamically allocate an int and assign its address to pointer

// The pointer gets a valid address with memory allocated

*p2 = 99;

cout<< p2 <<endl; // Print address after allocation

cout<< *p2 <<endl; // Print value point-to

delete p2;   // Remove the dynamically allocated storage

Observe that new and delete operators work on *pointer*.

To initialize the allocated memory, you can use an initializer for fundamental types, or invoke a constructor for an object. For example,

// use an initializer to initialize a fundamental type (such as int, double)

int * p1 = new int(88);

double * p2 = new double(1.23);

// C++11 *brace initialization* syntax

int * p1 = new int {88};

double * p2 = new double {1.23};

// invoke a constructor to initialize an object (such as Date, Time)

Date * date1 = new Date(1999, 1, 1);

Time * time1 = new Time(12, 34, 56);

You can dynamically allocate storage for *global* pointers inside a function. Dynamically allocated storage inside the function remains even after the function exits. For example,

```
// Dynamically allocate global pointers (TestDynamicAllocation.cpp)
#include <iostream>
using namespace std;

int * p1, * p2;  // Global int pointers

// This function allocates storage for the int*
// which is available outside the function
void allocate() {
  p1 = new int;   // Allocate memory, initial content unknown
  *p1 = 88;   // Assign value into location pointed to by pointer
  p2 = new int(99); // Allocate and initialize
}

Int main() {
allocate();
cout<< *p1 <<endl;  // 88
cout<< *p2 <<endl;  // 99
  delete p1; // Deallocate
  delete p2;
  return 0;
}
```

The main differences between static allocation and dynamic allocations are:

1. In static allocation, the compiler allocates and deallocates the storage automatically, and handle memory management. Whereas in dynamic allocation, you, as the programmer, handle the memory allocation and deallocation yourself (via new and delete operators). You have full control on the pointer addresses and their contents, as well as memory management.

2. Static allocated entities are manipulated through named variables. Dynamic allocated entities are handled through pointers.

3.

new[] and delete[] Operators

Dynamic array is allocated at runtime rather than compile-time, via the new[] operator. To remove the storage, you need to use the delete[] operator (instead of simply delete). For example,

```
/* Test dynamic allocation of array  (TestDynamicArray.cpp) */
#include <iostream>
#include <cstdlib>
using namespace std;

intmain() {
constint SIZE = 5;
int * pArray;

pArray = new int[SIZE];  // Allocate array via new[] operator

// Assign random numbers between 0 and 99
  for (inti = 0; i< SIZE; ++i) {
    *(pArray + i) = rand() % 100;
  }
// Print array
  for (inti = 0; i< SIZE; ++i) {
cout<< *(pArray + i) << " ";
  }
```

cout<<endl;

delete[] pArray; // Deallocate array via delete[] operator

  return 0;

}

C++03 does not allow your to initialize the dynamically-allocated array. C++11 does with the brace initialization, as follows:

// C++11

int * p = new int[5] {1, 2, 3, 4, 5};

Pointer, Array and Function

Array is Treated as Pointer

In C/C++, an array's name is a pointer, pointing to the first element (index 0) of the array. For example, suppose that numbers is an int array, numbers is a also an int pointer, pointing at the first element of the array. That is, numbers is the same as &numbers[0]. Consequently, *numbers is number[0]; *(numbers+i) is numbers[i].

For example,

/* Pointer and Array (TestPointerArray.cpp) */

#include <iostream>

using namespace std;

intmain() {

constint SIZE = 5;

int numbers[SIZE] = {11, 22, 44, 21, 41}; // An int array

// The array name numbers is an int pointer, pointing at the

// first item of the array, i.e., numbers = &numbers[0]

cout<<&numbers[0] <<endl; // Print address of first element (0x22fef8)

cout<< numbers <<endl;   // Same as above (0x22fef8)

cout<< *numbers <<endl;   // Same as numbers[0] (11)

cout<< *(numbers + 1) <<endl;   // Same as numbers[1] (22)

cout<< *(numbers + 4) <<endl;   // Same as numbers[4] (41)

}

4.2 Pointer Arithmetic

As seen from the previous section, if numbers is an int array, it is treated as an int pointer pointing to the first element of the array. (numbers + 1) points to the next int, instead of having the next sequential address. Take note that an int typically has 4 bytes. That is (numbers + 1) increases the address by 4, or sizeof(int). For example,

intnumbers[] = {11 22, 33};

int * iPtr = numbers;

cout<<iPtr<<endl;   // 0x22cd30

cout<<iPtr + 1 <<endl;   // 0x22cd34 (increase by 4 - sizeofint)

cout<< *iPtr<<endl;   // 11

cout<< *(iPtr + 1) <<endl; // 22

cout<< *iPtr + 1 <<endl;   // 12

USE OF MALLOC, CALLOC AND FREE FUNCTIONS

Functions malloc, calloc, realloc and free are used to allocate /deallocate memory on heap in C/C++ language. These functions should be used with great caution to avoid memory leaks and dangling pointers.

malloc (Allocating uninitialized memory)

This functions allocates the memory and returns the pointer to the allocated memory. Signature of function is

void *malloc(size_t size);

size_t corresponds to the integral data type returned by the language operator sizeof and is used to represent the size (in bytes) of an object. It is defined (In string.h header in C language andheader in C++) as an unsigned integral type. It is just an indication that the type is used to hold number of memory bytes (and not usual unsigned int).

The below code allocates memory for 10 integers and assign the address of allocated memory (address of the first byte of memory) to int pointer ptr

int * ptr = (int*) malloc(10 * sizeof(int));

- If the system is not able to allocate the requested memory on heap then malloc returns NULL.
- If size is zero (malloc(0)), then malloc returns either a NULL pointer or a valid pointer which can be passed to free function for successful memory dealocation. The actual value depends on the implementation.
- malloc returns a void pointer which need to casted to appropriate type before dereferencing. (The way we typecasted it to int* above.
- Memory returned by malloc is not initialized an holds garbage value.

Because malloc can return a NULL pointer in case it is not able to allocate memory, The value returned is first checked against valid memory allocation before using the pointer.

int * ptr = (int*) malloc(10 * sizeof(int));

if(ptr == NULL)

   // Unable to allocate memory. Take Action.

else

   // Memory allocation successful. can use ptr

While specifying the size absolute values should be avoided to make the code platform independent. For example: If you know that your compiler (plus machine) allocates 2 bytes to integers and you want to allocate memory for one integer, than also you should NOT write code like

int* ptr = (int*) malloc(2); // BAD CODE.. hardcode value 2

because this code will fail when will compile it on 4-byte machines (which allocates 4-bytes to integers). A good way is to always use sizeof operator

int* ptr = (int*) malloc(sizeof(int)); // BAD CODE.. hardcode value 2

It will get the actual size of int on machine and pass that value to malloc. Such mistakes are only committed by programmers new to C language.

calloc (Memory allocation + Initialization)

Calloc also allocates memory on heap like malloc does. The only difference is that calloc also initialize the memory with zero (malloc returns uninitialized memory).

Signature of calloc is:

void* calloc(size_tnum, size_t size);

It will allocate memory to an array of num elements where each element is of size bytes. we need to pass two parameters to calloc because it need to assign zero to each elements (hence it need to know how many elements are there).

The below code will allocate memory of one integer on heap, initialize it with zero and store the address in pointer ptr.

int * ptr = (int*) calloc(1, sizeof(int));

Except for initialization part, everything we wrote about malloc is true about calloc also.

Due to the alignment requirements, the number of allocated bytes is not necessarily equal to (num*size). This is typical to the struct where individual fields are alligned to word boundaries.

Realloc (change the size of memory block on heap)

Suppose a pointer (say ptr) is pointing to a memory of 10 int on heap allocated using malloc as below.

int * ptr = (int*)malloc(10*sizeof(int));

You want to increase the size of memory pointed to by ptr from 10 to 20, without loosing the contents of already allocated memory. In this case you can call the realloc function. Signature of realloc is:

void *realloc(void *ptr, size_t size);

where ptr is the pointer to the previously (currently) allocated block of memory and size is the new size (in bytes) for the new memory block.

It is possible that the function will move the memory block to a new location because it is not able to allocate memory just after the existing memory block as shown in the picture below:
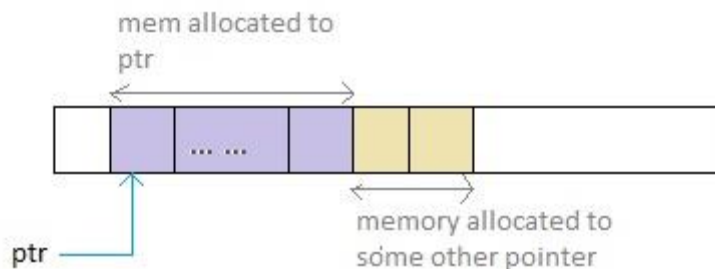
in this case realloc will allocate memory for 20 integers somewhere else and then copy the contents of first 10 locations from here to the new place. will deallocate the existing memory and return a pointer to the new memory. Hence ptr will change.

Programmer, should be aware with this fact because it may result in dangling pointers. consider the case below:

int * ptr1 = (int*) malloc(5 * sizeof(int));

int * ptr2 = ptr1;

ptr2 = (int*) realloc(ptr2, 10 * sizeof(int));

// ptr1 may become a dangling pointer

Int this case both ptr1 and ptr2 are pointing to the same memory location.

When realloc is called, the memory location pointed to by both the pointers may get deallocated (in case the contiguous space is not available just after the memory block). ptr2 will now point to the newly shifted location on the heap (returned by realloc), but ptr1 is still pointing to the old location (which is now deallocated).

Hence, ptr1 is a dangling pointer.

- If pointer passed to realloc is null, then it will behave exactly like malloc.
- If the size passed is zero, and ptr is not NULL then the call is equivalent to free.
- If the area is moved to new location then a free on the previous location is called.
- If contents will not change in the existing region. The new memory (in case you are increasing memory in realloc) will not be initialized and will hold garbage value.
- If realloc() fails the original block is left untouched; it is not freed or moved.

Free (Free the memory allocated using malloc, calloc or realloc)

free functions frees the memory on the heap, pointed to by a pointer. Signature of free function is void free(void* ptr);

- ptr must be pointing to a memory which is allocated using malloc, calloc or realloc.
- If ptr is called on a memory which is not on heap or on a dangling pointer, then the behavior is undefined.
- If ptr is NULL, then free does nothing and returns (So, its ok to call free on null pointers).

int x = 2;

int* ptr = &x;

free(ptr); //UNDEFINED.

int *ptr2; // UN initialized, hence dangling pointer

free(ptr2); //UNDEFINED

int *ptr3 = NULL;

free(ptr3); //OK.

Crashes in malloc(), calloc(), realloc(), or free() are almost always related to heap corruption, such as overflowing an allocated chunk or freeing the same pointer twice.

Use of new and delete

new/delete

- Allocate/release memory
    1. Memory allocated from 'Free Store'
    2. Returns a fully typed pointer.
    3. new (standard version) never returns a NULL (will throw on failure)
    4. Are called with Type-ID (compiler calculates the size)
    5. Has a version explicitly to handle arrays.
    6. Reallocating (to get more space) not handled intuitively (because of copy constructor).
    7. Whether they call malloc/free is implementation defined.
    8. Can add a new memory allocator to deal with low memory (set_new_handler)
    9. operator new/delete can be overridden legally
    10. constructor/destructor used to initialize/destroy the object

malloc/free

- Allocates/release memory
    1. Memory allocated from 'Heap'
    2. Returns a void*
    3. Returns NULL on failure
    4. Must specify the size required in bytes.
    5. Allocating array requires manual calculation of space.
    6. Reallocating larger chunk of memory simple (No copy constructor to worry about)
    7. They will **NOT** call new/delete
    8. No way to splice user code into the allocation sequence to help with low memory.
    9. malloc/free can **NOT** be overridden legally

Table comparison of the features:

Feature	new/delete	malloc/free
Memory allocated from	'Free Store'	'Heap'
Returns	Fully typed pointer	void*
On failure	Throws (never returns NULL)	Returns NULL
Required size	Calculated by compiler	Must be specified in bytes
Handling arrays	Has an explicit version	Requires manual calculations
Reallocating	Not handled intuitively	Simple (no copy constructor)
Call of reverse	Implementation defined	No
Low memory cases	Can add a new memory allocator	Not handled by user code
Overridable	Yes	No
Use of (con-)/destructor	Yes	No

Technically memory allocated by new comes from the 'Free Store' while memory allocated by malloc comes from the 'Heap'. Whether these two areas are the same is an implementation details, which is another reason that malloc and new can not be mixed.

C++ Opening and Closing Files

In C++, you open a file, you must first obtain a stream. There are the following three types of streams:

input

output

input/output

Create an Input Stream

To create an input stream, you must declare the stream to be of class ifstream. Here is the syntax:

ifstream fin;

Create an Output Stream

To create an output stream, you must declare it as class ofstream. Here is an example:

ofstreamfout;

Create both Input/Output Streams

Streams that will be performing both input and output operations must be declared as class fstream. Here is an example:

fstreamfio;

Opening a File in C++

Once a stream has been created, next step is to associate a file with it. And thereafter the file is available (opened) for processing.

Opening of files can be achieved in the following two ways :

Using the constructor function of the stream class.

Using the function open().

The first method is preferred when a single file is used with a stream. However, for managing multiple files with the same stream, the second method is preferred. Let's discuss each of these methods one by one.

Opening File Using Constructors

We know that a constructor of class initializes an object of its class when it (the object) is being created. Same way, the constructors of stream classes (ifstream, ofstream, or fstream) are used to initialize file stream objects with the filenames passed to them. This is carried out as explained here:

To open a file named myfile as an input file (i.e., data will be need from it and no other operation like writing or modifying would take place on the file), we shall create a file stream object of input type i.e., ifstream type. Here is an example:

ifstreamfin("myfile", ios::in) ;

The above given statement creates an object, fin, of input file stream. The object name is a user-defined name (i.e., any valid identifier name can be given). After creating the ifstream object fin, the file myfile is opened and attached to the input stream, fin. Now, both the data being read from myfile has been channelised through the input stream object.

Now to read from this file, this stream object will be used using the getfrom operator (">>"). Here is an example:

char ch;

fin >>ch ;   // read a character from the file

float amt ;

fin >>amt ;   // read a floating-point number form the file

Similarly, when you want a program to write a file i.e., to open an output file (on which no operation can take place except writing only). This will be accomplish by

creating ofstream object to manage the output stream

associating that object with a particular file

Here is an example,

ofstreamfout("secret" ios::out) ;   // create ofstream object named as fout

This would create an output stream, object named as fout and attach the file secret with it.

Now, to write something to it, you can use << (put to operator) in familiar way. Here is an example,

int code = 2193 ;

fout<< code << "xyz" ;   /* will write value of code

                           and "xyz" to fout's associated

                           file namely "secret" here. */

The connections with a file are closed automatically when the input and the output stream objects expires i.e., when they go out of scope. (For example, a global object expires when the program terminates). Also, you can close a connection with a file explicitly by using the close() method :

fin.close() ;   // close input connection to file

fout.close() ;   // close output connection to file

Closing such a connection does not eliminate the stream; it just disconnects it from the file. The stream still remains there. For example, after the above statements, the streams fin and fout still exist along with the buffers they manage. You can reconnect the stream to the same file or to another file, if required. Closing a file flushes the buffer which means the data remaining in the buffer (input or output stream) is moved out of it in the direction it is ought to be. For example, when an input file's connection is closed, the data is moved from the input buffer to the program and when an output file's connection is closed, the data is moved from the output buffer to the disk file.

Opening Files Using Open() Function

There may be situations requiring a program to open more than one file. The strategy for opening multiple files depends upon how they will be used. If the situation requires simultaneous processing of two files, then you need to create a separate stream for each file. However, if the situation demands sequential processing of files (i.e., processing them one by one), then you can open a single stream and associate it with each file in turn. To use this approach, declare a stream object without initializing it, then use a second statement to associate the stream with a file. For example,

```
ifstreamfin;                    // create an input stream
fin.open("Master.dat", ios::in);    // associate fin stream with file Master.dat
:                               // process Master.dat
fin.close();                    // terminate association with Master.dat

fin.open("Tran.dat", ios::in);      // associate fin stream with file Tran.dat
:                               // process Tran.dat
fin.close();                    // terminate association
```

The above code lets you handle reading two files in succession. Note that the first file is closed before opening the second one. This is necessary because a stream can be connected to only one file at a time.

The Concept of File Modes

The filemode describes how a file is to be used : to read from it, to write to it, to append it, and so on.

When you associate a stream with a file, either by initializing a file stream object with a file name or by using the open() method, you can provide a second argument specifying the file mode, as mentioned below :

stream_object.open("filename", (filemode)) ;

The second method argument of open(), the filemode, is of type int, and you can choose one from several constants defined in the ios class.

List of File Modes in C++

Following table lists the filemodes available in C++ with their meaning :

Constant	Meaning	Stream Type
ios :: in	It opens file for reading, i.e., in input mode.	ifstream
ios :: out	It opens file for writing, i.e., in output mode. This also opens the file in ios :: trunc mode, by default. This means an existing file is truncated when opened, i.e., its previous contents are discarded.	ofstream
ios :: ate	This seeks to end-of-file upon opening of the file. I/O operations can still occur anywhere within the file.	ofstream ifstream
ios :: app	This causes all output to that file to be appended to the end. This value can be used only with files capable of output.	ofstream
ios :: trunc	This value causes the contents of a pre-existing file by the same name	ofstream

	to be destroyed and truncates the file to zero length.	
ios :: nocreate	This cause the open() function to fail if the file does not already exist. It will not create a new file with that name.	ofstream
ios :: noreplace	This causes the open() function to fail if the file already exists. This is used when you want to create a new file and at the same time.	ofstream
ios :: binary	This causes a file to be opened in binary mode. By default, files are opened in text mode. When a file is opened in text mode, various character translations may take place, such as the conversion of carriage-return into newlines. However, no such character translations occur in file opened in binary mode.	ofstream ifstream

However, no such character translations occur in file opened in binary mode.

ofstream

ifstream

If the ifstream and ofstream constructors and the open() methods take two arguments each, how have we got by using just one in the previous examples ?

As you probably have guessed, the prototypes for these class member functions provide default values for the second argument (the filemode argument). For example, the ifstreamopen() method and constructor use ios :: in (open for reading) as the default value for the mode argument, while the ofstream open() method and constructor use ios :: out (open for writing) as the default.

The fstream class does not provide a mode by default and, therefore, one must specify the mode explicitly when using an object of fstream class.

Both ios::ate and ios::app place you at the end of the file just opened. The difference between the two is that the ios::app mode allows you to add data to the end of the file only, when the ios::ate mode lets you write data anywhere in the file, even over old data.

You can combine two or more filemode constants using the C++ bitwise OR operator (symbol |). For example, the following statement :

ofstreamfout;
fout.open("Master", ios :: app | ios :: nocreate);
will open a file in the append mode if the file exists and will abandon the file opening operation if the file does not exist.

To open a binary file, you need to specify ios :: binary along with the file mode, e.g.,

fout.open("Master", ios :: app | ios :: binary);
or,

fout.open("Main", ios :: out | ios :: nocreate | ios :: binary);
Closing a File in C++

As already mentioned, a file is closed by disconnecting it with the stream it is associated with. The close() function accomplishes this task and it takes the following general form :

stream_object.close();
For example, if a file Master is connected with an ofstream object fout, its connections with the stream fout can be terminated by the following statement :

fout.close() ;

C++ Opening and Closing a File Example

Here is an example given, for the complete understanding on:

how to open a file in C++ ?

how to close a file in C++ ?

Let's look at this program.


```
/* C++ Opening and Closing a File
 * This program demonstrates, how
 * to open a file to store or retrieve
 * information to/from it. And then how
 * to close that file after storing
 * or retrieving the information to/from it. */



#include<conio.h>
#include<string.h>
#include<stdio.h>
#include<fstream.h>
#include<stdlib.h>
void main()
{
        ofstreamfout;
        ifstream fin;
        char fname[20];
        char rec[80], ch;
        clrscr();
        cout<<"Enter file name: ";
        cin.get(fname, 20);
```

```
fout.open(fname, ios::out);
if(!fout)
{
        cout<<"Error in opening the file "<<fname;
        getch();
        exit(1);
}
cin.get(ch);

cout<<"\nEnter a line to store in the file:\n";
cin.get(rec, 80);
fout<<rec<<"\n";
cout<<"\nThe entered line stored in the file successfully..!!";
cout<<"\nPress any key to see...\n";
getch();
fout.close();
fin.open(fname, ios::in);
if(!fin)
{
        cout<<"Error in opening the file "<<fname;
        cout<<"\nPress any key to exit...";
        getch();
        exit(2);
}

cin.get(ch);
fin.get(rec, 80);
cout<<"\nThe file contains:\n";
cout<<rec;
cout<<"\n\nPress any key to exit...\n";
```

```
        fin.close();

        getch();

}
```

Here is the sample run of the above C++ program:


```
C:\TURBOC~1\Disk\TurboC3\SOURCE\1.EXE

Enter file name: myfile.txt

Enter a line to store in the file:
How to open and close a file in C++. This is the line.

The entered line stored in the file successfully..!!
Press any key to see...

The file contains:
How to open and close a file in C++. This is the line.

Press any key to exit...
```
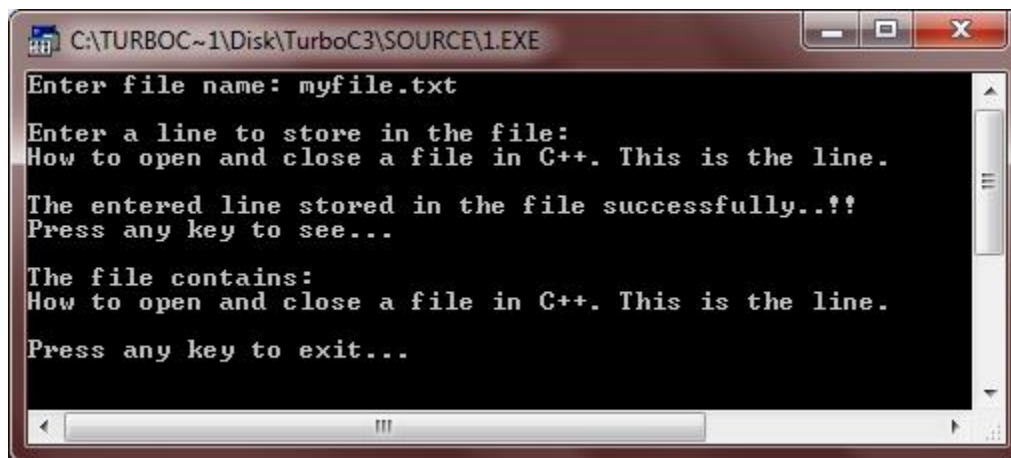
Read File in C++

To read a file in C++ programming, you have to first open that file using the function open() and then start reading the file's content as shown here in the following program.

C++ Programming Code to Read File

First make a textual file named "filename.txt" in your BIN (for TurboC++ user) folder present inside TurboC++ directory, to open this file for reading.

Following C++ program opens a file named filename.txt to read the content present inside this file, if there is an error in opening a file then program puts a message on the screen for the error, and if the file will be read then it will display the file (content of the file) but this program limits to only one line of the file which is to be read. To know more, go to 2nd next program, which will clear your doubt, that program will read a file and display the contents of it. For now, go through the following program which will read a file and display its content on the screen:

/* C++ Program - Read a File */

```
#include<iostream.h>
#include<conio.h>
#include<stdlib.h>
#include<stdio.h>
#include<fstream.h>
void main()
{
        clrscr();
        char c[1000];
        ifstreamifile;
        ifile.open("filename.txt") ;
        if(!ifile)
        {
                cout<<"Error in opening file..!!";
                getch();
                exit(1);
        }
        cout<<"Data in file = ";
        while(ifile.eof()==0)
        {
                ifile>>c;
                cout<<c<<" ";
        }
        ifile.close();
        getch();
}
```

Write To File in C++

To write some content in a file using C++ programming, you have to enter the file name with extension to open that file using the function open(), then after opening the desired file, again ask to the user to enter some content (some line of text) to store in the file. And at last, close the file after use using the function close() as shown in the following program.
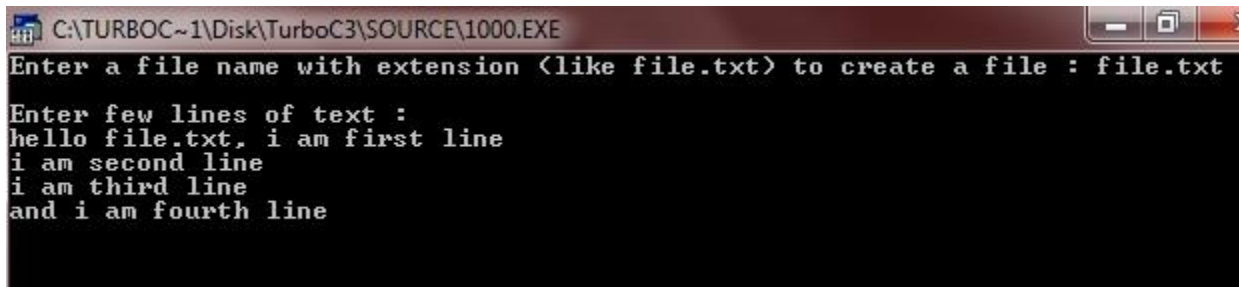
C++ Programming Code to Write Content to File

Following C++ program ask to the user to enter file name to open (if file present inside the directory) or create (if file not present inside the directory), then ask to the user to enter some line of text to store these lines inside the files for further use :

```
/* C++ Program - Write to File */
#include<iostream.h>
#include<conio.h>
#include<string.h>
#include<stdio.h>
#include<fstream.h>
#include<stdlib.h>
void main()
{
        clrscr();
        ofstreamfp;
        char s[100], fname[20];
        cout<<"Enter a file name with extension (like file.txt) to create a file : ";
        gets(fname);
        fp.open(fname);
        if(!fp)
        {
                cout<<"Error in opening file..!!";
                getch();
                exit(1);
```

```
        }
        cout<<"Enter few lines of text :\n";
        while(strlen(gets(s))>0)
        {
                fp<<s;
                fp<<"\n";
        }
        fp.close();
        getch();
}
```

When the above C++ program is compile and executed, it will produce the following result:

C++ program to write content to file

Here after writing four line of text that is :

hello file.txt, i am first line

i am second line

i am third line

and i am fourth line

After writing the above four line, you will press double ENTER key (line break), your all the four line will be written in the file named file.txt and output screen backed to the source code.

Random access files

Random Access of Files (File Pointers)

Using file streams, we can randomly access binary files. By random access, you can go to any position in the file as you wish (instead of going in a sequential order from the first character to the last). Technically this bookmarker is a file pointer and it determines as to where to write the next character (or from where to read the next character). We have seen that file streams can be created for input (ifstream) or for output (ofstream). For ifstream the pointer is called as 'get' pointer and for ofstream the pointer is called as 'put' pointer. fstream can perform both input and output operations and hence it has one 'get' pointer and one 'put' pointer. The 'get' pointer indicates the byte number in the file from where the next input has to occur. The 'put' pointer indicates the byte number in the file where the next output has to be made. There are two functions to enable you move these pointers in a file wherever you want to:

seekg( ) - belongs to the ifstream class
seekp( ) - belongs to the ofstream class

We'll write a program to copy the string "Hi this is a test file" into a file called mydoc.txt. Then we'll attempt to read the file starting from the 8th character (using the seekg( ) function).

Strings are character arrays terminated in a null character ('\0'). If you want to copy a string of text into a character array, you should make use of the function:

strcpy (character-array, text);

to copy the text into the character array (even blank spaces will be copied into the character array). To make use of this function you might need to include the string.h header file.

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

```
intmain( )
{
ofstream out("c:/mydoc.txt",ios::binary);
char text[80];
strcpy(text,"Hi this is a test file");
out<<text;
out.close( );
ifstream in("c:/mydoc.txt",ios::binary);
in.seekg(8);
cout<<endl<<"Starting from position 8 the contents are:"<<endl;

while ( !in.eof( ) )
{
char ch;
in.get(ch);
if ( !in.eof( ) )
   {
cout<<ch;
   }
}

in.close( );
return 0;
}
```

The output is:

Starting from position 8 the contents are:

is a test file

As you can see, the output doesn't display, "Hi this " because they are the first 7 characters present in the file. We've asked the program to display from the 8th character onwards using the seekg( ) function.

in.seekg(8);

will effectively move the bookmarker to the 8th position in the file. So when you read the file, you will start reading from the 8th position onwards.

The following fragment of code is interesting:

```
while ( !in.eof( ) )
{
char ch;
in.get(ch);
if ( !in.eof( ) )
   {
cout<<ch;
   }
}
```

You might be wondering as to why we need to check for the EOF again using an 'if' statement. To understand the reason, try the program by removing the 'if' statement. The result will be surprising and interesting. Think over it and you will be able to figure out the logic.

The syntax for seekg( ) or seekp( ) is:

seekg(position, ios::beg)

seekg(position, ios::cur)

seekg(position, ios::end)

By default (i.e. if you don't specify 'beg' or 'cur' or 'end') the compiler will assume it as ios::beg.

ios::beg – means that the compiler will count the position from the beginning of the file.

ios::cur – means the compiler starts counting from the current position.

ios::end – it will move the bookmarker starting from the end of the file.

Just like we have 2 functions to move the bookmarker to different places in the file, we have another 2 functions that can be used to get the present position of the bookmarker in the file.

For input streams we have: tellg( )

For output streams we have :tellp( )

You would think that the value returned by tellg( ) and tellp ( ) are integers. They are like integers but they aren't. The actual syntax for these functions will be:

streampostellg( );

where streampos is an integer value that is defined in the compiler (it is actually a typedef).

Of course you can say:

int position = tellg( );

Now, the variable 'position' will have the location of the bookmarker. But you can also say:

streampos position = tellg( );

This will also give the same result. 'streampos' is defined internally by the compiler specifically for file-streams.

Similarly, the syntax of seekg( ) and seekp ( ) was mentioned as:

seekg(position, ios::beg)

Again in the above syntax, 'position' is actually of type 'streampos'.

Sequential and Random Access Files

Basically variables are used for temporary storage and files are used for permanent storage of data. Based on how files are accessed, they can be divided into sequential and random access files. Actually this division of files depends on how we read and write to files (physically the file is stored as a sequence of bytes in memory).

Random access files overcome this problem since they have fixed length records. The problem here is that even if we want to store a small sized record we still have to occupy the entire fixed record length. This leads to wastage of some memory space. For example if we are using 10

bytes to store a complete sentence in the file then even if you want to store a single letter (like 'a') 10 bytes will also be used up for this. But even though some memory space is wasted this method will speed up access time (because now we know where each record is stored. If a record length is fixed as 10 bytes, then the fifth record will start at byte number 50 and it is easier to jump directly to that location instead of reading the first four records before accessing the fifth).

Word processing program usually store files in a sequential format while database management programs store files in a random access format. A simple real life analogy: Audio tapes are accessed sequentially while audio CDs (Compact Discs) are accessed randomly.

So, how do we create sequential and random access files in C++? Actually we have already covered both these topics without explicitly using the terms sequential and random access. Whenever you make use of the 'read' and 'write' functions to write structures/objects to a file, you are actually creating a random access file (because every record will have the size of the structure). Whenever you use the << and >> operator to read and write to disk files, you are accessing the file sequentially (this was the first example program). Whenever you write to a stream (or a file) using << operator, you are writing varying length records to the file. For example: You might first write a string of 10 characters followed by an integer. Then you may write another string of 20 characters followed by a 'double'. Thus the records are all of varying lengths.

To effectively use random access files we make use of the seekp( ) and seekg ( ) functions. Though these can be used on sequential files it will not be very useful in sequential access files (because when you are searching for a data you are forced to read each and every character/byte, whereas in random access files you can jump to the particular record that you are interested in).

Command Line Arguments

You know that functions can have arguments. You also know that main ( ) is a function. In this section we'll take a look at how to pass arguments to the main function. Usually filenames are passed to the program.

First of all, let us suppose that we have a file by the name marks.cpp. From this file we make an exe file called marks.exe. This is an executable file and you can run it from the command prompt. The command prompt specifies what drive and directory you are currently in. The command prompt can be seen as the ms-dos prompt.

C:\WINDOWS>

This denotes that you are currently in C drive and in the directory named Windows. (By the way, if you want to go to the MS DOS command prompt from Windows, just go to "Start" and click on "Run". Type "command" in the text box and click "Ok").

Your marks.exe program is in this directory (let us assume it is here. If it isn't in this directory then you have to change to that particular directory). To run the program you will type:

C:\WINDOWS> marks name result

You must be thinking that we will type only the name of the program? In this case the C++ program that you wrote is assumed to have arguments for the main ( )function (i.e. in the marks.cpp file you have provided arguments for the main( ) function):

int main (intargc, char * argv[ ])

argc (the first argument - argument counter) stands for the number of arguments passed from the command line.

argv (argument vector) is an array of character type that points to the command line arguments.

In our example, the value of 'argc' is 3 (marks, name, result). Hence for 'argv' we have an array of 3 elements. They are:

argv[0] which is marks.

argv[1] which is name

argv[2] which is result

Note: argv[0] will be the name that invokes the program (i.e. it is the name of the program that you have written).

If you feel a little vague in this section don't worry. In the next section we'll take a look at a simple program.

A program using Command Line Arguments

```cpp
// This file is named test.cpp
#include <iostream.h>
int main ( intargc, char * argv[ ] )
{
cout<<"The value of argument counter (argc) is: "<<argc;
inti;
for ( i = 0 ; i<argc ; i ++ )
{
cout<<endl<<argv[i];
}
return 0;
}
```

Save the file as test.cpp. Compile it and then make the executable file (test.exe). If you run test.exe from Windows (i.e. by just double clicking on the file),

the output will be as follows:

The value of argument counter (argc) is: 1

c:\windows\test.exe

This will be the output since you didn't specify the arguments. To specify the arguments you have to go to DOS prompt. From there type:

c:\windows>test one two three

You have to go to the folder in which you have the test.exe file (I assume that your program is in the windows directory in C drive).

The output will be:

The value of argument counter (argc) is: 4
c:\windows\t.exe
one
two
three

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 is 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 byreplacement. This replacement can be an expression, a statement, a block or simply anything. The preprocessor does not understand C++ proper, it simply replaces any occurrence of identifier by replacement.

```
#define TABLE_SIZE 100
int table1[TABLE_SIZE];
int table2[TABLE_SIZE];
```

After the preprocessor has replaced TABLE_SIZE, the code becomes equivalent to:

```
int table1[100];
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, exactly as you would expect if it was a function:

// function macro
#include <iostream>
usingnamespace std;
 	
#define getmax(a,b) ((a)>(b)?(a):(b))
 	
intmain()
{
int x=5, y;
  y= getmax(x,2);
cout<< y <<endl;
cout<<getmax(7,x) <<endl;
return 0;
}
	5
	7

Defined macros are not affected by block structure. A macro lasts until it is undefined with the #undef preprocessor directive:

```
#define TABLE_SIZE 100
int table1[TABLE_SIZE];
#undef TABLE_SIZE
#define TABLE_SIZE 200
int table2[TABLE_SIZE];
```

This would generate the same code as:

```
int table1[100];
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):

```
#define str(x) #x
cout<<str(test);
```

This would be translated into:

```
cout<<"test";
```

The operator ## concatenates two arguments leaving no blank spaces between them:

```
#define glue(a,b) a ## b
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++.

Conditional inclusions (#ifdef, #ifndef, #if, #endif, #else and #elif)

These directives allow to include or discard part of the code of a program if a certain condition is met.

#ifdef allows a section of a program to be compiled only if the macro that is specified as the parameter has been defined, no matter which its value is. For example:

```
#ifdef TABLE_SIZE
int table[TABLE_SIZE];
#endif
```

In this case, the line of code int table[TABLE_SIZE]; is only compiled if TABLE_SIZE was previously defined with #define, independently of its value. If it was not defined, that line will not be included in the program compilation.

#ifndef serves for the exact opposite: the code between #ifndef and #endif directives is only compiled if the specified identifier has not been previously defined. For example:

```
#ifndef TABLE_SIZE
#define TABLE_SIZE 100
#endif
int table[TABLE_SIZE];
```

In this case, if when arriving at this piece of code, the TABLE_SIZE macro has not been defined yet, it would be defined to a value of 100. If it already existed it would keep its previous value since the #define directive would not be executed.

The #if, #else and #elif (i.e., "else if") directives serve to specify some condition to be met in order for the portion of code they surround to be compiled. The condition that follows #if or #elif can only evaluate constant expressions, including macro expressions. For example:

```
#if TABLE_SIZE>200
#undef TABLE_SIZE
#define TABLE_SIZE 200


#elif TABLE_SIZE<50
#undef TABLE_SIZE
#define TABLE_SIZE 50


#else
#undef TABLE_SIZE
#define TABLE_SIZE 100
#endif


int table[TABLE_SIZE];
```

Notice how the entire structure of #if, #elif and #else chained directives ends with #endif. The behavior of #ifdef and #ifndef can also be achieved by using the special operators defined and !defined respectively in any #if or #elif directive:

```
#if defined ARRAY_SIZE
#define TABLE_SIZE ARRAY_SIZE
#elif !defined BUFFER_SIZE
#define TABLE_SIZE 128
#else
#define TABLE_SIZE BUFFER_SIZE
#endif
```

Line control (#line)

When we compile a program and some error happens during the compiling process, the compiler shows an error message with references to the name of the file where the error happened and a line number, so it is easier to find the code generating the error.

The #line directive allows us to control both things, the line numbers within the code files as well as the file name that we want that appears when an error takes place. Its format is:

#line number "filename"

Where number is the new line number that will be assigned to the next code line. The line numbers of successive lines will be increased one by one from this point on.

"filename" is an optional parameter that allows to redefine the file name that will be shown. For example:

```
#line 20 "assigning variable"
int a?;
```

This code will generate an error that will be shown as error in file "assigning variable", line 20.

Error directive (#error)

This directive aborts the compilation process when it is found, generating a compilation error that can be specified as its parameter:

```
#ifndef __cplusplus
#error A C++ compiler is required!
#endif
```

This example aborts the compilation process if the macro name __cplusplus is not defined (this macro name is defined by default in all C++ compilers).

Source file inclusion (#include)

This directive has been used assiduously in other sections of this tutorial. When the preprocessor finds an #includedirective it replaces it by the entire content of the specified header or file. There are two ways to use #include:

1 #include <header>
2 #include "file"

In the first case, a *header* is specified between angle-brackets <>. This is used to include headers provided by the implementation, such as the headers that compose the standard library (iostream, string,...). Whether the headers are actually files or exist in some other form is *implementation-defined*, but in any case they shall be properly included with this directive.

The syntax used in the second #include uses quotes, and includes a *file*. The *file* is searched for in an *implementation-defined* manner, which generally includes the current path. In the case that the file is not found, the compiler interprets the directive as a *header* inclusion, just as if the quotes ("") were replaced by angle-brackets (<>).

Pragma directive (#pragma)

This directive is used to specify diverse options to the compiler. These options are specific for the platform and the compiler you use. Consult the manual or the reference of your compiler for more information on the possible parameters that you can define with #pragma.

If the compiler does not support a specific argument for #pragma, it is ignored - no syntax error is generated.

Predefined macro names

The following macro names are always defined (they all begin and end with two underscore characters, _):

macro	value
__LINE__	Integer value representing the current line in the source code file being compiled.

__FILE__	A string literal containing the presumed name of the source file being compiled.
__DATE__	A string literal in the form "Mmmddyyyy" containing the date in which the compilation process began.
__TIME__	A string literal in the form "hh:mm:ss" containing the time at which the compilation process began.
__cplusplus	An integer value. All C++ compilers have this constant defined to some value. Its value depends on the version of the standard supported by the compiler: • **199711L**: ISO C++ 1998/2003 • **201103L**: ISO C++ 2011 Non conforming compilers define this constant as some value at most five digits long. Note that many compilers are not fully conforming and thus will have this constant defined as neither of the values above.
__STDC_HOSTED__	1 if the implementation is a *hosted implementation* (with all standard headers available) 0 otherwise.

The following macros are optionally defined, generally depending on whether a feature is available:

macro	value
__STDC__	In C: if defined to 1, the implementation conforms to the C standard. In C++: Implementation defined.
__STDC_VERSION__	In C: • **199401L**: ISO C 1990, Ammendment 1 • **199901L**: ISO C 1999 • **201112L**: ISO C 2011

	In C++: Implementation defined.
__STDC_MB_MIGHT_NEQ_WC__	1 if multibyte encoding might give a character a different value in character literals
__STDC_ISO_10646__	A value in the form yyyymmL, specifying the date of the Unicode standard followed by the encoding of wchar_t characters
__STDCPP_STRICT_POINTER_SAFETY__	1 if the implementation has *strict pointer safety* (see get_pointer_safety)
__STDCPP_THREADS__	1 if the program can have more than one thread

Particular implementations may define additional constants.

For example:

// standard macro names
#include <iostream>
usingnamespace std;
 	
intmain()
{
cout<<"This is the line number "<< __LINE__;
cout<<" of file "<< __FILE__ <<".\n";
cout<<"Its compilation began "<< __DATE__;
cout<<" at "<< __TIME__ <<".\n";
cout<<"The compiler gives a __cplusplus value of "<< __cplusplus;
return 0;}
	This is the line number 7 of file /home/jay/stdmacronames.cpp.
	Its compilation began Nov 1 2005 at 10:12:29.
	The compiler gives a __cplusplus value of 1

POSSIBLE QUESTIONS – UNIT IV

Part-A

Online Examinations   (One marks)

1. The wrapping up of data & function into a single unit is known as _____
   a) Polymorphism   **b) encapsulation**   c) functions   d) data members

2. _____statement is frequently used to terminate the loop in the switch case()
   a) jump   b) goto   c) continue   **d) break**

3. What will you use if you are not intended to get a return value?
   a) static   b) const   c) volatile   **d) void**

4. Where the default values of parameter have to be specified?
   a) Function call   b) Function definition **c)Function prototype** d) Both B or C

5. Where does the return statement returns the execution of the program?
   a) main function   b) **caller function**   **c)** same function   d) none

6. What does a reference provide?
   a) Alternate name for the class   **b) Alternate name for the variable**
   c) Alternate name for the pointer   d) none

7. The changes made in the members of a structure are available in the calling function if
   a) pointer to structure is passed as argument   b) structure variable is passed
   c) the member other then pointer type are passed as argument   d) both option a and c

8. For accessing a structure element using a pointer, you must use?
   a) Pointer operator (&)   b) Dot operators(.)
   c) Pointer operator(*)   **d) Arrow operator(->)**

9. The ---------- function reads character input into the variable line
   a) getline()   b)line()   c) gets()   d) None.

10. File streams act as an _____ between programs and files.
    a) interface   b)converter   c) translator   d) operator

Part-B   2 MARKS

1. Define calloc( ).
2. Define malloc( ).
3. What is memory allocation in c++ define its types.
4. What is dynamic memory allocation in c++?
5. Write about new and delete operators with example.
6. What is preprocessor?
7. Define macros.
8. What are file stream classes?
9. How will you open a file give example?
10. Define put( ), get( ), read( ) and write( ) with example.
11. What is a file?
12. What is the use of file inclusion?

Part-C   6 MARKS

1. With proper example explain malloc.
2. Write note on random access files in c++.
3. With proper example explain calloc in c++
4. Explain with example i) #include   ii) #error
5. Write a program that swaps two numbers.
6. Discuss Macros with example program.
7. Write a program to display Fibonacci series using recursion.
8. Discuss the role of opening a file use of ifstream and ofstream header files.
9. With proper example explain new and delete operators in c++.
10. Explain reading and writing text files.
11. Write a program to create a file for employee pay slip and manipulate it?
12. Write a program to copy the content of a file into another using command line
13. Write in detail about conditional compilation using preprocessors

KARPAGAM ACADEMY OF HIGHER EDUCATION

COIMBATORE - 21

DEPARTMENT OF COMPUTER SCIENCE,CA & IT

CLASS : I B.Sc COMPUTER SCIENCE

BATCH : 2017-2020

Part -A Online Examinations   **(1 mark questions)**

SUBJECT: PROGRAMMING FUNDAMENTALS USING   SUBJECT CODE: 17CSU101
C/C++

UNIT-IV

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	none	Console and file
5	The data accepted with default setting by I/O function of the language is known as	Formatted	Unformatted	Argumented	None.	Unformatted
6	_____ is used as the input stream to read data.	Cout	Printf	Cin	Scanf	Cin

#	Question					
7	cin and cout are _____ for input and output of data.	user defined stream	system defined stream	Pre defined stream	none	system defined stream
8	The data obtained or represented with some manipulators are called _____.	formatted data	unformatted data	extracted data	None.	formatted data
9	The manipulator << endl is equivalent to____	'\t'	'\r'	'\n'	'\b'	'\n'
10	Precision() is an _____ format function	Manipulator	Istream	ios	user defined	ios
11	Width of the output field is set using the _____	width()	iomanip.h	Manipulator	None	width()
12	Stream and stream classes are used to implement its I/O operations with the _____	the console and disk files	cin and cout	manipulators	none	the console and disk files
13	The interface supplied by an I/O system which is independent of actual device is called _____	stream	class	object	none.	stream
14	A _____ is a sequence of bytes.	Stream	class	object	none	Stream
15	The _____ streams automatically open when the program begins its execution	user defined	predefined	input	output	predefined
16	The class that is defined to various streams to deal with both the console and disk files is called _____	stream class	derived class	object	none	stream class
17	____ provide an interface to physical devices through buffers.	stream buffer	iostream	ostream	istream	stream buffer

		>> and <<	+ and −	* and &&	− and .	>> and <<
18	The _____ are called as overloaded operators					
19	The >> operator is overloaded in the _____	istream	ostream	iostream	None	istream
20	The ____ functions are used to handle the single character I/O operation.	get() and put()	clrscr() and getch()	cin and cout	None	get() and put()
21	____ 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()	none	getline() and write()
22	The getline() reads character input to the _____ line	datatype	function	variable	none	variable
23	_____ is used to clear the flags specified.	width()	precision()	setf()	unsetf()	unsetf()
24	_____ is used to specify the required field size for displaying an output value	width()	self	fill()	none	width()
25	By default the floating numbers are printed with _____ after the decimal point.	5 digits	6	7	8	6
26	____ returns the setting in effect until it is reset	width	precision()	setf()	fill()	precision()
27	A _____ is a collection of related data stored in a particular area on a disk.	Field	File	Row	Vector	File
28	File streams act as an _____ between programs and files.	interface	converter	translator	operator	interface

#	Question					
29	_____ is to set the file buffer to read and write.	filebuf	filestream	thread	package	filebuf
30	_____ inherits get(), getline(), read(), seekg(), and tellg() from istream.	conio	ifstream	fstream	iostream	ifstream
31	Put(), seekp(), tellp(), and write() functions are inherited by ofstream from _____	ostream	fstream	ifstream	istream	ostream
32	_____ inherits all functions from istream and ostream through iostream	file stream	ofstream	fstream	ifstream	fstream
33	The eof ( ) stands for _____.	end of file	error opening file	error of file	none of the above	end of file
34	Command line arguments are used with _____ function	main()	member function	with all function	none of the above	main()
35	Feof function is used to test	End of file condition	Beginning of file condition	Middle of the file condition	Previous file position	End of file condition
36	_____ is a another memory allocation function th	Malloc()	Realloc()	Calloc()	Free()	Calloc()
37	With the dynamic run time allocation it is responsible to	Malloc()	Realloc()	Calloc()	Free()	Free()
38	List , queue and stack are all inherently	One dimensional	Two dimensional	Multi-dimensional	Hierarchal	One dimensional
39	Program that processes the source code before it passes	Preprocessor	Function	Library function	structure function	Preprocessor

#	Question					
40	C preprocessor offers a special feature known as	Unconditional compilation	Debugging statement	Macro compilation	Conditional compilation	Conditional compilation
41	FILE is a	Function	Structure	Defined data type	I/O function	Defined data type
42	Getc() is used for	Write a character	Read a character	Append a character	None of the above	Read a character
43	Fseek() is used for	Gives current position	Gives previous position	Sets the position to the beginning	Sets desired point	Sets desired point
44	Putw() is used for	Write a integer	Read a character	Append a character	None of the above	Write a integer
45	FILE is a structure defined in---	Not defined in I/O library	I/O library	Input library	output library	I/O library
46	Filename specified in FILE concept should have	Primary name and optional	Secondary name and optional	Only Primary name	Only optional period	Primary name and optional period
47	W mode is used for	Reading and writing	Only reading	Only writing	none	Only writing
48	Filename and mode should be specified in	Double quotation	Single quotation	With tilde symbol	None	Double quotation
49	Fprintf and fscanf function is used for	For printing and reading in the file	Only for reading	Only for writing	Scanning the variables	For printing and reading in the file

KARPAGAM ACADEMY OF HIGHER EDUCATION

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

Coimbatore - 641021.

(For the candidates admitted from 2017 onwards)

DEPARTMENT OF COMPUTER SCIENCE,CA & IT

SUBJECT : **PROGRAMMING FUNDAMENTALS USING C/C++**
SEMESTER : **I**   **L T P C**
SUBJECT CODE: **17CSU101**   **CLASS**   : **I B.Sc.CS**   **4 0 0 4**

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 Access, 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. Herbtz Schildt, 2003,*C++: The Complete Reference*, 4th Edition, McGraw Hill.

2. Bjarne Stroustrup, 2013, *The C++ Programming Language,* 4th Edition, Addison-Wesley.

3. Bjarne Stroustroup, 2014, *Programming - Principles and Practice using C++,* 2nd Edition, Addison-Wesley.

4. E Balaguruswamy, 2008,*Object Oriented Programming with C++,* 2nd Edition ,Tata McGraw-Hill Education.

5. Paul Deitel, Harvey Deitel, 2011*,C++ How to Program*, 8th Edition, Prentice Hall.

6. John R. Hubbard, 2000, *Programming with C++,* 2nd Edition ,Schaum's Series.

7. Stefan Bjornander, 2016,*C++ Windows Programming,* Published by Packt Publishing Ltd.

8. Scott Meyers, 2005, *Effective C++,* 3rd Edition, Published by Addison-Wesley.

9. Harry, H. Chaudhary, 2014 *,Head First C++ Programming: The Definitive Beginner's Guide*, First Create space Inc, 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*, 5th Edition, Published by Addison-Wesley.

12. Debasish Jana , 2014,*C++ And Object-Oriented Programming Paradigm*,Published by PHI Learning Pvt. Ltd.

13. Richard L. Stegman, 2016, *Focus on Object-oriented Programming With C++,*6th Edition ,CreateSpace Independent Publishing Platform,.

14. Andrew Koeni, Barbara, E. Moo,2000,*Accelerated using C++,* Published by Addison-Wesley .

WEBSITES

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.   http://www.cplusplus.com/

6.   http://www.cppreference.com/

UNIT - V

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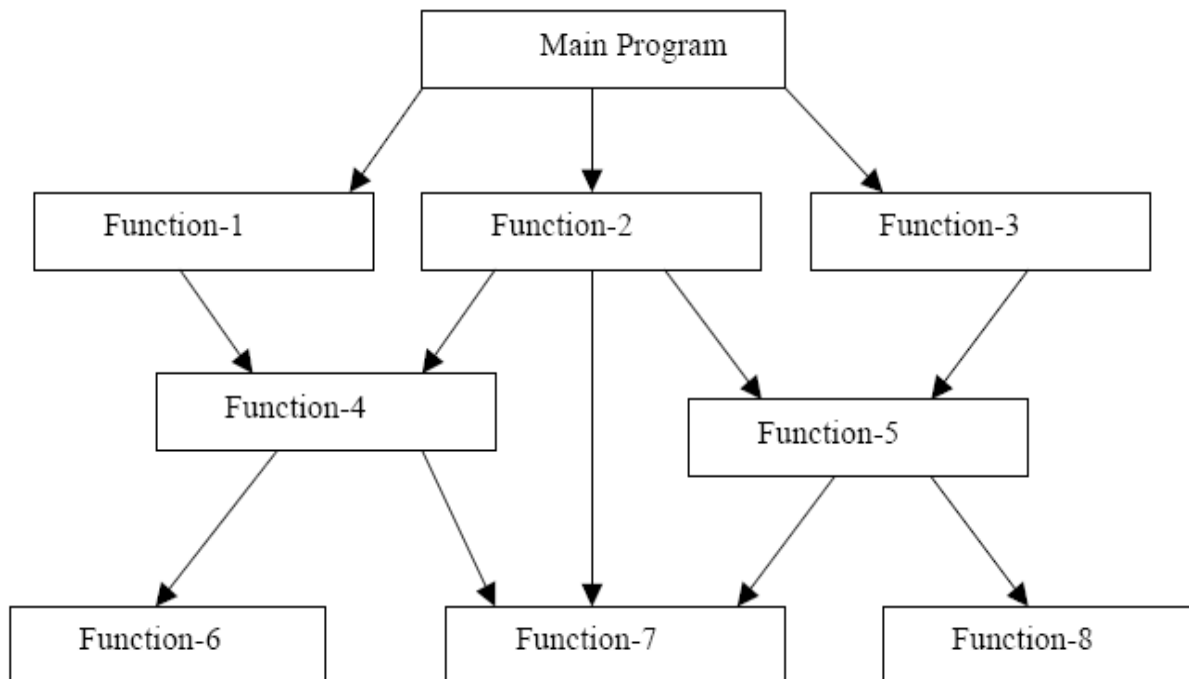
Procedure oriented programming basically consists of writing a list of instructions for the computer to follow, and organizing these instructions into groups known as *functions*. We normally use flowcharts to organize these actions and represent the flow of control from one action to another.

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
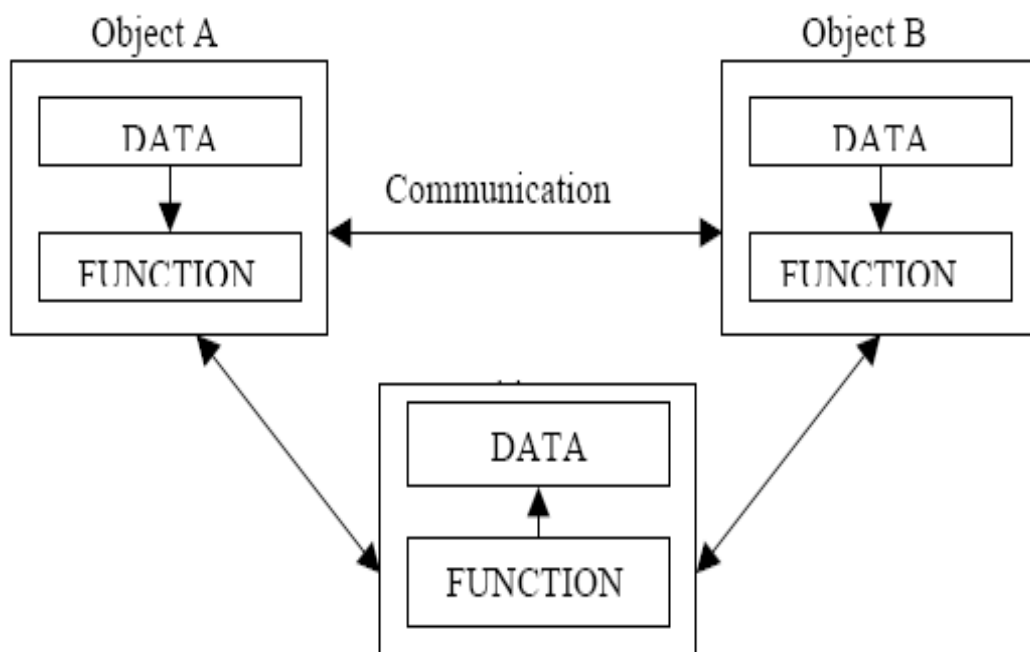
Some of the features of object oriented programming are:

• 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 ties 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 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 he 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 classifieds into different types, they are

- **Single level inheritance**
- **Multi-level inheritance**
- **Hybrid inheritance**
- **Hierarchial 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 exiting 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.

History of C++

C++ was written by Bjarne Stroustrup at Bell Labs during 1983-1985. C++ is an extension of C. Prior to 1983, Bjarne Stroustrup added features to C and formed what he called "C with Classes". He had combined the Simula's use of classes and object-oriented features with the power and efficiency of C. The term C++ was first used in 1983.

C And C++ Difference

- C does not have any classes or objects. It is procedure and function driven. There is no concept of access through objects and structures are the only place where there is a access through a compacted variable. c++ is object oriented.

- C structures have a different behaviour compared to c++ structures. Structures in c do not accept functions as their parts.

- C input/output is based on library and the processes are carried out by including functions. C++ i/o is made through console commands cin and cout.

- C functions do not support overloading. Operator overloading is a process in which the same function has two or more different behaviours based on the data input by the user.

- C does not support new or delete commands. The memory operations to free or allocate memory in c are carried out by malloc() and free().

- Undeclared functions in c++ are not allowed. The function has to have a prototype defined before the main() before use in c++ although in c the functions can be declared at the point of use.

- After declaring structures and enumerators in c we cannot declare the variable for the structure right after the end of the structure as in c++.

- For an int main() in c++ we may not write a return statement but the return is mandatory in c if we are using int main().

- In C++ identifiers are not allowed to contain two or more consecutive underscores in any position. C identifiers cannot start with two or more consecutive underscores, but may contain them in other positions.

- C has a top down approach whereas c++ has a bottom up approach.

- In c a character constant is automatically elevated to an integer whereas in c++ this is not the case.

- In c declaring the global variable several times is allowed but this is not allowed in c++.

Applications and Benefits of using OOP

Applications of using OOP:

- User interface design such as windows, menu ,…

- Real Time Systems

- Simulation and Modeling

- Object oriented databases

- AI and Expert System

- Neural Networks and parallel programming

- Decision support and office automation system etc

Benefits of OOP

- It is easy to model a real system as real objects are represented by programming objects in OOP. The objects are processed by their member data and functions. It is easy to analyze the user requirements.

- With the help of inheritance, we can reuse the existing class to derive a new class such that the redundant code is eliminated and the use of existing class is extended. This saves time and cost of program.

- In OOP, data can be made private to a class such that only member functions of the class can access the data. This principle of data hiding helps the programmer to build a secure program that can not be invaded by code in other part of the program.

- With the help of polymorphism, the same function or same operator can be used for different purposes. This helps to manage software complexity easily.

- Large problems can be reduced to smaller and more manageable problems. It is easy to partition the work in a project based on objects.

- It is possible to have multiple instances of an object to co-exist without any interference i.e. each object has its own separate member data and function.

- OOP provides a clear modular structure for programs.

- It is good for defining abstract data types.

- Implementation details are hidden from other modules and other modules has a clearly defined interface.

- It is easy to maintain and modify existing code as new objects can be created with small differences to existing ones.

- objects, methods, instance, message passing, inheritance are some important properties provided by these particular languages

- encapsulation, polymorphism, abstraction are also counts in these fundamentals of programming language.

- It implements real life scenario.

- In OOP, programmer not only defines data types but also deals with operations applied for data structures.

- More reliable software development is possible.

- Enhanced form of c programming language.

- The most important Feature is that it's procedural and object oriented nature.

- Much suitable for large projects.

- Fairly efficient languages.

- It has the feature of memory management.

Structure of 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 -
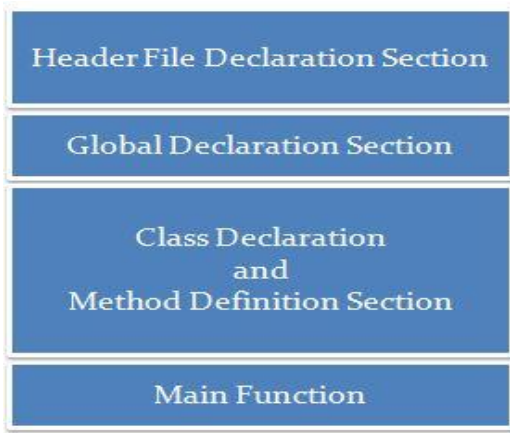
Section 1 : Header File Declaration Section

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 -

    o   Declaring Structure

    o   Declaring Class

    o   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.

// my first program in C++

#include <iostream.h>

int main ()

```
{
  cout << "Hello World!";
  return 0;
}
```

Hello World!

• The C++ language defines several headers, which contain information that is either necessary or useful to your program. For this program, the header **<iostream>** is needed.

• The line **using namespace std;** tells the compiler to use the std namespace. Namespaces are a relatively recent addition to C++.

• The next line **// main() is where program execution begins.** is a single-line comment available in C++. Single-line comments begin with // and stop at the end of the line.

• The line **int main()** is the main function where program execution begins.

• The next line **cout << "This is my first C++ program.";** causes the message "This is my first C++ program" to be displayed on the screen.

• The next line **return 0;** terminates main( )function and causes it to return the value 0 to the calling process.

The program has been structured in different lines in order to be more readable, but it is not compulsory to do so. For example, instead of

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

we could have written:

int main () { cout << " Hello World "; return 0; }in just one line and this would have had exactly the same meaning.

Comments.

Comments are pieces of source code discarded from the code by the compiler. They do nothing. Their purpose is only to allow the programmer to insert notes or descriptions embedded within the source code.

C++ supports two ways to insert comments:

// line comment

/* block comment */

Inline functions

C++ **inline** function is powerful concept that is commonly used with classes. If a function is inline, the compiler places a copy of the code of that function at each point where the function is called at compile time.

Any change to an inline function could require all clients of the function to be recompiled because compiler would need to replace all the code once again otherwise it will continue with old functionality.

To inline a function, place the keyword **inline** before the function name and define the function before any calls are made to the function. The compiler can ignore the inline qualifier in case defined function is more than a line.

A function definition in a class definition is an inline function definition, even without the use of the **inline**specifier.

Following is an example, which makes use of inline function to return max of two numbers:

```
#include <iostream>
 using namespace std;
inline int Max(int x, int y)
{
   return (x > y)? x : y;
}
// Main function for the program
int main( )
{
   cout << "Max (20,10): " << Max(20,10) << endl;
   cout << "Max (0,200): " << Max(0,200) << endl;
   cout << "Max (100,1010): " << Max(100,1010) << endl;
   return 0;
}
```

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

```
Max (20,10): 20
Max (0,200): 200
Max (100,1010): 1010
```

Inline function is the optimization technique used by the compilers. One can simply prepend inline keyword to function prototype to make a function inline. Inline function instruct compiler to insert complete body of the function wherever that function got used in code.

Advantages:-

1) It does not require function calling overhead.

2) It also save overhead of variables push/pop on the stack, while function calling.

3) It also save overhead of return call from a function.

4) It increases locality of reference by utilizing instruction cache.

5) After in-lining compiler can also apply intraprocedural optimization if specified. This is the most important one, in this way compiler can now focus on dead code elimination, can give more stress on branch prediction, induction variable elimination etc..

Disadvantages:-

1) May increase function size so that it may not fit on the cache, causing lots of cache miss.

2) After in-lining function if variables number which are going to use register increases than they may create overhead on register variable resource utilization.

3) It may cause compilation overhead as if some body changes code inside inline function than all calling location will also be compiled.

4) If used in header file, it will make your header file size large and may also make it unreadable.

5) If somebody used too many inline function resultant in a larger code size than it may cause thrashing in memory. More and more number of page fault bringing down your program performance.

6) Its not useful for embedded system where large binary size is not preferred at all due to memory size constraints.

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

Specifying a Class

The mechanism that allows you to combine data and the function in a single unit is called a class. Once a class is defined, you can declare variables of that type. A class variable is called object or instance. In other words, a class would be the data type, and an object would be the variable. Classes are generally declared using the keyword class, with the following format:

```
class class_name
{
  private:
    members1;
  protected:
```

```
    members2;
  public:
    members3;
};
```

Where class_name is a valid identifier for the class. The body of the declaration can contain members, that can be either data or function declarations, The members of a class are classified into three categories: private, public, and protected. Private, protected, and public are reserved words and are called member access specifiers. These specifiers modify the access rights that the members following them acquire.

- **private members** of a class are accessible only from within other members of the same class. You cannot access it outside of the class.

- **protected members** are accessible from members of their same class and also from members of their derived classes.

- Finally, **public members** are accessible from anywhere where the object is visible.

By default, all members of a class declared with the class keyword have private access for all its members. Therefore, any member that is declared before one other class specifier automatically has private access.

Here is a complete example :

```
class student
{
  private :
    int rollno;
    float marks;
  public:
    void getdata()
    {
      cout<<"Enter Roll Number : ";
      cin>>rollno;
      cout<<"Enter Marks : ";
      cin>>marks;
    }
    void displaydata()
    {
      cout<<"Roll number : "<<rollno<<"\nMarks : "<<marks;
```

}

};

Object Declaration

Once a class is defined, you can declare objects of that type. The syntax for declaring a object is the same as that for declaring any other variable. The following statements declare two objects of type student:

student st1, st2;

Accessing Class Members

Once an object of a class is declared, it can access the public members of the class.

st1.getdata();

Defining Member function of class

You can define Functions inside the class as shown in above example. Member functions defined inside a class this way are created as inline functions by default. It is also possible to declare a function within a class but define it elsewhere. Functions defined outside the class are not normally inline.

When we define a function outside the class we cannot reference them (directly) outside of the class. In order to reference these, we use the scope resolution operator, :: (double colon).

In this example, we are defining function getdata outside the class:

void student :: getdata()

{

   cout<<"Enter Roll Number : ";

   cin>>rollno;

   cout<<"Enter Marks : ";

   cin>>marks;

}

The following program demonstrates the general feature of classes. Member function initdata() is **defined inside the class**. Member functions getdata() and showdata() **defined outside the class**.

class student //specify a class

{

 private :

  int rollno; //class data members

  float marks;

 public:

```
    void init data(int r, int m)
    {
      rollno=r;
      marks=m;
    }
    void getdata(); //member function to get data from user
    void showdata();// member function to show data
};
void student :: getdata()
{
    cout<<"Enter Roll Number : ";
    cin>>rollno;
    cout<<"Enter Marks : ";
    cin>>marks;
}
void student :: showdata()
{
    cout<<"Roll number : "<<rollno<<"\nMarks : "<<marks;
}
int main()
{   student st1, st2; //define two objects of class student
    st1.initdata(5,78); //call member function to initialize
    st1.showdata();
    st2.getdata(); //call member function to input data
    st2.showdata(); //call member function to display data
    return 0;
}
```

Static data members

We can define class members static using **static** keyword. When we declare a member of a class as static it means no matter how many objects of the class are created, there is only one copy of the static member.

A static member is shared by all objects of the class. All static data is initialized to zero when the first object is created, if no other initialization is present. We can't put it in the class definition but it

can be initialized outside the class as done in the following example by redeclaring the static variable, using the scope resolution operator **::** to identify which class it belongs to.

Let us try the following example to understand the concept of static data members:

```
#include <iostream>
 using namespace std;
class Box
{   public:
      static int objectCount;
      // Constructor definition
      Box(double l=2.0, double b=2.0, double h=2.0)
      {
        cout <<"Constructor called." << endl;
        length = l;
        breadth = b;
        height = h;
        // Increase every time object is created
        objectCount++;
      }
      double Volume()
      {
        return length * breadth * height;
      }
   private:
      double length;    // Length of a box
      double breadth;   // Breadth of a box
      double height;    // Height of a box
};
// Initialize static member of class Box
int Box::objectCount = 0;
int main(void)
{   Box Box1(3.3, 1.2, 1.5);   // Declare box1
   Box Box2(8.5, 6.0, 2.0);   // Declare box2
   // Print total number of objects.
```

```
    cout << "Total objects: " << Box::objectCount << endl;
    return 0;
}
```

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

Constructor called.

Constructor called.

Total objects: 2

Static Member Function:

By declaring a function member as static, you make it independent of any particular object of the class. A static member function can be called even if no objects of the class exist and the **static** functions are accessed using only the class name and the scope resolution operator **::**.

A static member function can only access static data member, other static member functions and any other functions from outside the class.

Static member functions have a class scope and they do not have access to the **this** pointer of the class. You could use a static member function to determine whether some objects of the class have been created or not.

Let us try the following example to understand the concept of static function members:

```
#include <iostream>
 using namespace std;
class Box
{   public:
     static int objectCount;
     // Constructor definition
     Box(double l=2.0, double b=2.0, double h=2.0)
     {   cout <<"Constructor called." << endl;
       length = l;
       breadth = b;
       height = h;
       // Increase every time object is created
```

```
    objectCount++;
  }
  double Volume()
  {
    return length * breadth * height;
  }
  static int getCount()
  {      return objectCount;
  }
  private:
    double length;    // Length of a box
    double breadth;   // Breadth of a box
    double height;    // Height of a box
};
// Initialize static member of class Box
int Box::objectCount = 0;
int main(void)
{    // Print total number of objects before creating object.
  cout << "Inital Stage Count: " << Box::getCount() << endl;
  Box Box1(3.3, 1.2, 1.5);   // Declare box1
  Box Box2(8.5, 6.0, 2.0);   // Declare box2
  // Print total number of objects after creating object.
  cout << "Final Stage Count: " << Box::getCount() << endl;
  return 0;
}
```

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

Inital Stage Count: 0

Constructor called.

Constructor called.

Final Stage Count: 2

Example:

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

```
class stat
{
  int code;
  static int count;
  public:
  stat()
  {
   code=++count;
  }
  void showcode()
  {
   cout<<"\n\tObject number is :"<<code;
  }
  static void showcount()
  {        cout<<"\n\tCount Objects :"<<count;
  }
};
int stat::count;
 void main()
{   clrscr();
  stat obj1,obj2;
   obj1.showcount();
  obj1.showcode();
  obj2.showcount();
  obj2.showcode();
  getch();
}
```

Output:

Count Objects: 2

Object Number is: 1

Count Objects: 2

Object Number is: 2

Array of objects

Arrays of variables of type "class" are known as "**Array of objects**". The "identifier" used to refer the array of objects is a user defined data type.

```cpp
#include <iostream.h>
const int MAX  =100;
class Details
 {
  private:
    int salary;
    float roll;
  public:
    void getname( )
     { cout  << "\n Enter the Salary:";
       cin >> salary;
       cout  << "\n Enter the roll:";
       cin >> roll;
      }
    void putname( )
      {      cout << "Employees" << salary <<
       "and roll is" << roll << '\n';
      }
   };
 void main()
  { Details det[MAX];
   int n=0;
   char ans;
   do{
      cout  << "Enter the Employee Number::"  <<  n+1;
      det[n++].getname;
      cout << "Enter another (y/n)?: " ;
      cin >> ans;
    } while ( ans != 'n' );
   for (int j=0; j<n; j++)
    {
      cout << "\nEmployee Number is:: " << j+1;
```

        det[j].putname( );

     }   }

Result:

   Enter the Employee Number:: 1

   Enter the Salary:20

    Enter the roll:30

   Enter another (y/n)?: y

   Enter the Employee Number:: 2

   Enter the Salary:20

     Enter the roll:30

   Enter another (y/n)?: n

In the above example an array of object "det" is defined using the user defined data type "Details". The class element "getname()" is used to get the input that is stored in this array of objects and putname() is used to display the information.

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

Im 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 Constructer 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 intialized 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

Im Constructor

Values :10   20

Values :10   20

Simple Program for Copy Constructor Using C++ Programming

```cpp
#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

```cpp
#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; //defalut constructor invoked

  student st2(5,78); //parmeterized 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;

}
```

OPERATOR OVERLOADING & INHERITANCE

Defining Operator Overloading

One of the nice features of C++ is that you can give special meanings to operators, when they are used with user-defined classes. This is called *operator overloading*. You can implement C++ operator overloads by providing special member-functions on your classes that follow a particular naming convention. For example, to overload the + operator for your class, you would provide a member-function named operator+ on your class.

An operator is a symbol that indicates an operation. It is used to perform operation with constants and variables. Without an operator, programmer cannot build an expression.

The operator + (plus) can be used to perform addition of two variables, but the same is not applicable for objects. The compiler cannot perform addition of two objects. The compiler would throw an error if addition of two objects is carried out. Operator overloading helps programmer to use these operators with the objects of classes. The outcome of operator overloading is that objects can be used in a natural manner as the variables of basic data types.

The following set of operators is commonly overloaded for user-defined classes:

- = (assignment operator)
- + - * (binary arithmetic operators)
- += -=
- *= (compound assignment operators)
- == != (comparison operators)

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

Return type operator op(arglist)

{

Function body

}

Operator functions must be either member function or friend friend function

Friend function:

-One argument for unary operator and two for binary operator

Member function

- no argument for unary operator and one for binary operator
- object invoked implicitly

Invoked by expression such as

Op x or x op → unary operator

X op y → binary operator

RULES FOR OPERATOR OVERLOADING

1. Only existing operators can be overloaded. New operators can not be created.

2. The overloaded operator must have at least one operand that is of user defined data type.

3. We can't change the basic meaning of an operator. That is to say, we can't redefine the plus(+) operator to subtract one value from other.

4. Overloaded operators follow the syntax rules of the original operators. They can't be overridden.

5. There are some operators that can't be overloaded.

6. We can't use friend functions to overload certain operators. How-ever, member functions can be used to overload them.

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

8. Binary operators overloaded through a member function, take one explicit argument and those which are overloaded through a friend function take two explicit arguments. When using binary operators overloaded through a member function, the left hand operand must be an object of the relevant class.

9. Binary arithmetic operators such as +,-,* and / must explicitly return a value. They must not attempt to change their own arguments.

<u>Overloading unary operators</u>

The operator ++,-- and – are unary operators. The unary operators ++ and - - can be used as prefix or suffix with the functions. These operators have only single operand.

Example:

```
# include<iostream.h>
Class space
{
Int x;
Int y;
Int z;
Public :
Void getdata(int a, int b, int c) ;
Void display(void);
Void operator-();
};
```

```
Void space :: getdata(int a, int b, int c)

{

X=a;

Y=b;

Z=c;

}

Void space:: display()

{

Cout<<x<<" ";

Cout<<y<<" ";

Cout<<z<<"\n ";

}

Void space:: operator-()

{

X=-x ;

Y=-y;

Z=-z;

}

Void main()

{

Space s ;

s.getdata(10,-20,30) ;

cout<< " s : ";

s.display();

-s;

cout<< " s : ";

s.display();

}
```

Output:

S: 10 -20 30

S: -10 20 -30

Overloading binary operators

When a member operator function overloads a binary operator, the function will have only one parameter. This parameter will receive the object that is on the right side of the operator. The object on the left side is the object that generates the call to the operator function and is passed implicitly by **this pointer**.

Binary operators require two operands. Binary operators are overloaded by using member functions and friend functions.

The unary operators take two arguments and following are the examples of Binary operators. You use binary operators very frequently like addition (+) operator, subtraction (-) operator and division (/) operator.

If overloaded as a member function they require one argument. The argument contains value of the object. Which is to the right of the operator. If we want to perform the addition of two objects o1 and o2, the overloading function should be declared as follows:

Operator(num o2);

Where, num is a class name and o2 is an object.

To call function operator the statement is as follows:

O3=o1+o2;

We know that a member function can be called by using class of that object. Hence, the called member function is always preceded by the object. Here, in the above statement, the object o1 invokes the function operator() and the object o2 is used as an argument for the function. The above statement can also be written as follows:

O3=o1.operator+(o2);

Here the data members of o1 are passed directly and data members of o2 are passed as an argument. While overloading binary operators, the left-hand operand calls the operator function and righthand operand is used as an argument.

Example

PROGRAM:

```
#include<iostream.h>
#include<conio.h>
 class complex
{       int a,b;
   public:
         void getvalue()
         {
           cout<<"Enter the value of Complex Numbers a,b:";
```

```
        cin>>a>>b;
    }
    complex operator+(complex ob)
    {
            complex t;
            t.a=a+ob.a;
            t.b=b+ob.b;
            return(t);
    }
    complex operator-(complex ob)
    {       complex t;
            t.a=a-ob.a;
            t.b=b-ob.b;
            return(t);
    }
    void display()
    {
            cout<<a<<"+"<<b<<"i"<<"\n";
    }
};
void main()
{   clrscr();
  complex obj1,obj2,result,result1;
   obj1.getvalue();
  obj2.getvalue();
   result = obj1+obj2;
  result1=obj1-obj2;
   cout<<"Input Values:\n";
  obj1.display();
  obj2.display();
   cout<<"Result:";
  result.display();
 result1.display();
   getch();
```

}

Output:

Enter the value of Complex Numbers a, b

4   5

Enter the value of Complex Numbers a, b

2   2

Input Values

4 + 5i

2 + 2i

Result

6 +   7i

2 +   3i

Example 2:

```
#include<iostream.h>
class money
{
int rs;
int ps;
money()
{
rs=0;
ps=0;
}
money(int r,int p)
{
rs=r;
ps=p;
}
money operator +(money);
void display();
}
money money :: operator +(money m1)
{
money temp;
```

```
temp.ps=m1.ps+ps;

if(temp.ps>99)

{

temp.rs++;

temp.ps-=99

}

temp.rs+=m1.rs+rs;

return temp;

}

void money :: display()

{

cout<<"Total Rupees "<<rs;

cout<<"\n Total Paise "<<ps;

}

void main()

{

money m1(10,55);

money m2(11,55);

money m3;

m3=m1+m2;

m3.display();

}
```

Output of the Program

Total Rupees 22

Total Paise 10

Overloading binary operators using friend functions

The friend can be used alternatively with member functions for overloading of binary operators. the friend function requires two operands to be passed as arguments.

O3=o1+o2;

O3=operator + (o1,o2);

Both the above statements have the same meaning. In the second statement, two objects are passed to the operator function.

The use of member function and friend function produces the same result. Friend functions are useful when we require performing an operation with operand of two different types. Consider the statements

X=y+3;

X=3+y;

Where x and y are objects of same type. The first statement is valid. However, the second statement will not work. The first operand must be an object of the same class. This problem can be overcome by using friend function. The friend function can be called without using object. The friend function can be used with standard data type as left-hand operand and with an object as right-hand operand.

It is possible to overload an operator relative to a class by using a friend rather than a member function A friend function does not have a **this** pointer. In the case of a binary operator, this means that a friend operator function is passed both operands explicitly. For unary operators, the single operand is passed.

We cannot use a friend to overload the assignment operator. The assignment operator can be overloaded only by a member operator function.

Example

#include<iostream.h>

class myclass

{

 int num1;

  int num2;

public:

myclass()

{

 num1=0;

 num2=0;

}

myclass(int x1,int x2)

{

 num1=x1;

 num2=x2;

}

 void show()

 {

```
cout<<endl<<"score 1:";<<num1;
cout<<endl<<"score 2:";
}


friend myclass operator+(myclass objmyclass1,myclass objmyclass2);

friend myclass operator*(myclass objmyclass1,myclass objmyclass2);

};

myclass operator+(myclass objmyclass1,myclass objmyclass2)

{

 myclass temp;

temp.num1=objmyclass1.num1+objmyclass2.num1;

temp.num2=objmyclass1.num2+objmyclass2.num2;

return temp;

}

myclass operator*(myclass objmyclass1,myclass objmyclass2)

{

myclass temp;

temp.num1=objmyclass1.num1*objmyclass2.num1;

temp.num2=objmyclass1.num2*objmyclass2.num2;

return temp;

}

int main()

{

 myclass player1(10.20);

 player1.show();

 myclass player2(40.30);

 player2.show();

 myclass result;

 result=player1 + player2;

 result.show();

 result=player1 * player2;

 result.show();

 return 0;

}
```

OUTPUT:

score 1:10

score 2:20

score 1:40

score 2:30

score 1:50

score 2:50

score 1:400

score 2:600

Inheritance

The mechanism that allows us to extend the definition of a class without making any physical changes to the existing class is inheritance.

Inheritance lets you create new classes from existing class. Any new class that you create from an existing class is called **derived class**; existing class is called **base class**.

The inheritance relationship enables a derived class to inherit features from its base class. Furthermore, the derived class can add new features of its own. Therefore, rather than create completely new classes from scratch, you can take advantage of inheritance and reduce software complexity.

Forms of Inheritance

1. **Single Inheritance:** It is the inheritance hierarchy wherein one derived class inherits from one base class.

2. **Multiple Inheritance:** It is the inheritance hierarchy wherein one derived class inherits from multiple base class(es)

3. **Hierarchical Inheritance:** It is the inheritance hierarchy wherein multiple subclasses inherit from one base class.

4. **Multilevel Inheritance:** It is the inheritance hierarchy wherein subclass acts as a base class for other classes.

5. **Hybrid Inheritance:** The inheritance hierarchy that reflects any legal combination of other four types of inheritance.
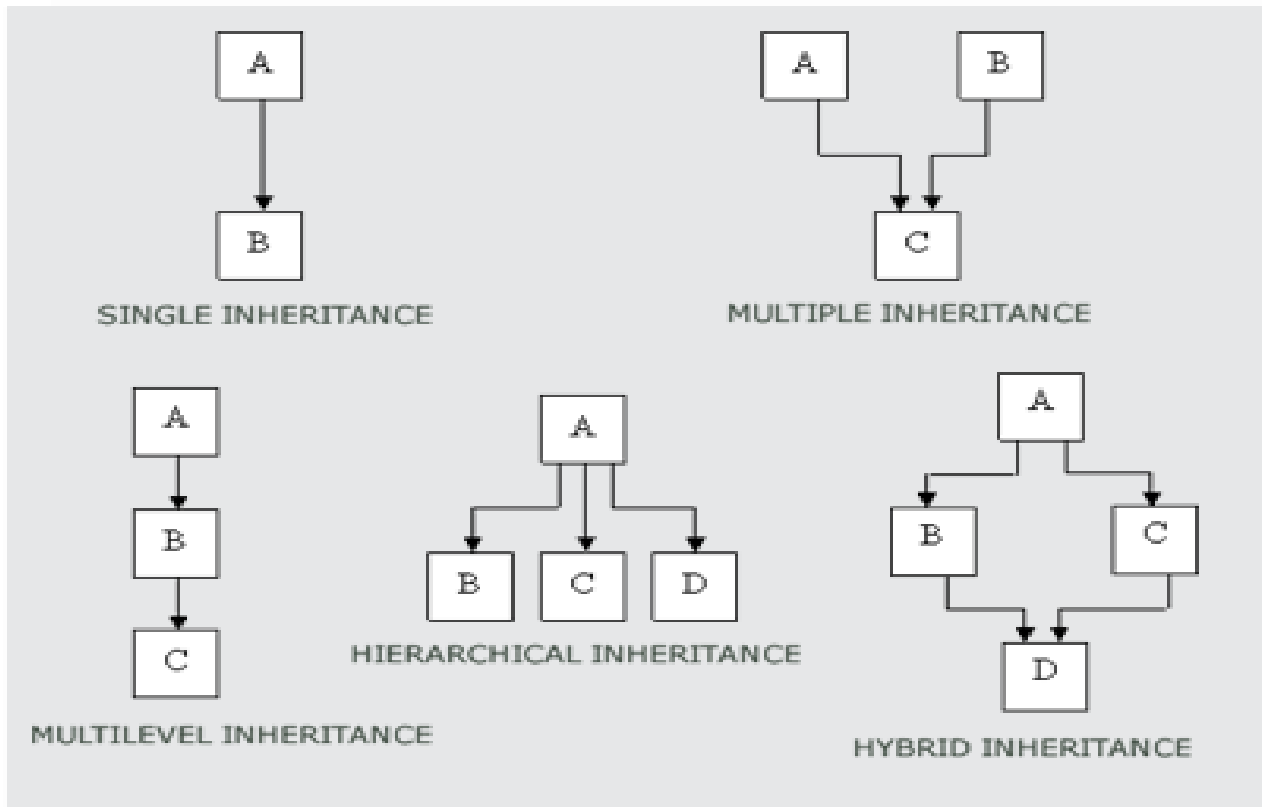
Defining Derived classes

A class that was created based on a previously existing class (i.e., base class). A derived class inherits all of the member variables and methods of the base class from which it is derived.

In order to derive a class from another, we use a colon (:) in the declaration of the derived class using the following format :

class derived_class: memberAccessSpecifier base_class

{

        ...

};

Where derived_class is the name of the derived class and base_class is the name of the class on which it is based. The member Access Specifier may be public, protected or private. This access specifier describes the access level for the members that are inherited from the base class.

Member Access Specifier	How Members of the Base Class Appear in the Derived Class
Private	Private members of the base class are inaccessible to the derived class.

	Protected members of the base class become private members of the derived class.
	Public members of the base class become private members of the derived class.
Protected	Private members of the base class are inaccessible to the derived class.
	Protected members of the base class become protected members of the derived class.
	Public members of the base class become protected members of the derived class.
Public	Private members of the base class are inaccessible to the derived class.
	Protected members of the base class become protected members of the derived class.
	Public members of the base class become public members of the derived class.

In principle, a derived class inherits every member of a base class except constructor and destructor. It means private members are also become members of derived class. But they are inaccessible by the members of derived class.

Following example further explains concept of inheritance:
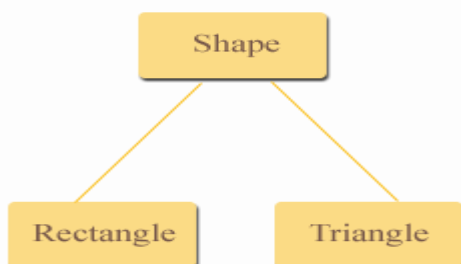

```
class Shape
{protected:
        float width, height;
public:
        void set_data (float a, float b)
        {       width = a;
                height = b;

        }
};
class Rectangle: public Shape
{public:
```

```
        float area ()
        {return (width * height);
        }
};
class Triangle: public Shape
{public:
        float area ()
        { return (width * height / 2);
        }
};
int main ()
{
        Rectangle rect;
        Triangle tri;
        rect.set_data (5,3);
        tri.set_data (2,5);
        cout << rect.area() << endl;
        cout << tri.area() << endl;
        return 0;
}
```

output :

15

5

The object of the class Rectangle contains:

Width, height inherited from Shape becomes the protected member of Rectangle.

set_data() inherited from Shape becomes the public member of Rectangle

area is Rectangle's own public member

 The object of the class Triangle contains:

Width, height inherited from Shape becomes the protected member of Triangle.

set_data() inherited from Shape becomes the public member of Triangle

area is Triangle's own public member

set_data () and area() are public members of derived class and can be accessed from outside class i.e.

from main()

Single Inheritance

In "single inheritance," a common form of inheritance, classes have only one base class. Single inheritance enables a derived class to inherit properties and behavior from a single parent class. It allows a derived class to inherit the properties and behavior of a base class, thus enabling code reusability as well as adding new features to the existing code. This makes the code much more elegant and less repetitive. Inheritance is one of the key features of object-oriented programming Single inheritance is safer than multiple inheritance if it is approached in the right way. It also enables a derived class to call the parent class implementation for a specific method if this method is overridden in the derived class or the parent class constructor.

PROGRAM:PAYROLL SYSTEM USING SINGLE INHERITANCE

```cpp
#include<iostream.h>
#include<conio.h>
 class emp
{   public:
    int eno;
    char name[20],des[20];
    void get()
    {   cout<<"Enter the employee number:";
        cin>>eno;
        cout<<"Enter the employee name:";
        cin>>name;
        cout<<"Enter the designation:";
        cin>>des;
    }
};
 class salary:public emp
{    float bp,hra,da,pf,np;
  public:
    void get1()
    {    cout<<"Enter the basic pay:";
        cin>>bp;
        cout<<"Enter the Humen Resource Allowance:";
        cin>>hra;
        cout<<"Enter the Dearness Allowance :";
```

```
          cin>>da;
          cout<<"Enter the Profitablity Fund:";
          cin>>pf;
     }
     void calculate()
     {    np=bp+hra+da-pf;
     }
     void display()
 {
 cout<<eno<<"\t"<<name<<"\t"<<des<<"\t"<<bp<<"\t"<<hra<<"\t"<<da<<"\t"<<pf<<"\t"<<np<<"\
n";
     }
};
 void main()
{    int i,n;
   char ch;
   salary s[10];
   clrscr();
   cout<<"Enter the number of employee:";
   cin>>n;
   for(i=0;i<n;i++)
   {      s[i].get();
          s[i].get1();
          s[i].calculate();
   }
   cout<<"\ne_no \t e_name\t des \t bp \t hra \t da \t pf \t np \n";
   for(i=0;i<n;i++)
   {
          s[i].display();
   }
   getch();
}
```

Output:

Enter the Number of employee:1

Enter the employee No: 150

Enter the employee Name: ram

Enter the designation: Manager

Enter the basic pay: 5000

Enter the HR allowance: 1000

Enter the Dearness allowance: 500

Enter the profitability Fund: 300

E.No	E.name	des	BP	HRA	DA	PF	NP
150	ram	Manager	5000	1000	500	300	6200

Multilevel Inheritance

Multilevel Inheritance is a method where a derived class is derived from another derived class.

```cpp
#include <iostream.h>
class mm
 {
  protected:
   int rollno;
  public:
   void get_num(int a)
    { rollno = a; }
   void put_num()
    { cout << "Roll Number Is:\n"<< rollno << "\n"; }
 };
class marks : public mm
 {
  protected:
   int sub1;
   int sub2;
  public:
   void get_marks(int x,int y)
    {
     sub1 = x;
     sub2 = y;
    }
```

```cpp
    void put_marks(void)
     {
      cout << "Subject 1:" << sub1 << "\n";
      cout << "Subject 2:" << sub2 << "\n";
     }
   };
 class res : public marks
   {
    protected:
     float tot;
    public:
     void disp(void)
       {
        tot = sub1+sub2;
        put_num();
        put_marks();
        cout << "Total:"<< tot;
       }
   };
 int main()
   {
    res std1;
    std1.get_num(5);
    std1.get_marks(10,20);
    std1.disp();
    return 0;
   }
```

Result:

Roll Number Is:

5

Subject 1: 10

Subject 2: 20

Total: 30

Multiple Inheritance

1. Deriving directly from more than one class is usually called multiple inheritance. Since it's widely believed that this concept complicates the design and debuggers can have a hard time with it, multiple inheritance can be a controversial topic. class can inherit behaviors and features from more than one superclass. This contrasts with single inheritance, where a class may inherit from at most one superclass.

```cpp
#include<iostream.h>
#include<conio.h>
 class student
{
   protected:
     int rno,m1,m2;
   public:
         void get()
       {
               cout<<"Enter the Roll no :";
               cin>>rno;
               cout<<"Enter the two marks   :";
               cin>>m1>>m2;
       }
};
class sports
{   protected:
     int sm;              // sm = Sports mark
   public:
         void getsm()
       {
        cout<<"\nEnter the sports mark :";
        cin>>sm;


       }
};
class statement:public student,public sports
```

```
{    int tot,avg;
  public:
   void display()
        {    tot=(m1+m2+sm);
          avg=tot/3;
        cout<<"\n\n\tRoll No    : "<<rno<<"\n\tTotal      : "<<tot;
         cout<<"\n\tAverage    : "<<avg;
         }
};
void main()
{   clrscr();
  statement obj;
  obj.get();
  obj.getsm();
  obj.display();
  getch();
}
```

Output:

        Enter the Roll no: 100

         Enter two marks

         90

         80

         Enter the Sports Mark: 90

         Roll No: 100

        Total   : 260

        Average: 86.66

Hierarchical Inheritance

Hierarchical Inheritance is a method of inheritance where one or more derived classes is derived from common base class.

Example

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

```
    {  protected:
       int l;
     public:
       void set_values (int x)
         { l=x;}
    };
class Square: public Side
  {  public:
       int sq()
       { return (l *l); }
    };
class Cube:public Side
  { public:
      int cub()
       { return (l *l*l); }
    };
int main ()
  {  Square s;
    s.set_values (10);
    cout << "The square value is::" << s.sq() << endl;
    Cube c;
    c.set_values (20);
    cout << "The cube value is::" << c.cub() << endl;
    return 0;
  }
```

Result:

The square value is:: 100

The cube value is::8000

Hybrid Inheritance

"Hybrid Inheritance" is a method where one or more types of inheritance are combined together and used.

Example
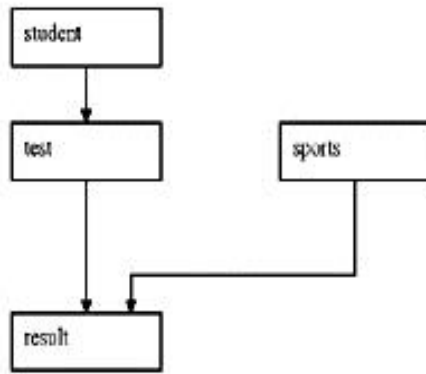
```
#include <iostream>
using namespace std;
class student
{        protected:
            int roll_no;
         public:
               void get_no(int a)
               {
                     roll_no=a;
               }
               void put_no(void)
               {
               cout << "Roll No:"<<roll_no<<"\n";
               }
    };
class test : public student
{      protected:
            float part1,part2;
       public:
            void get_marks(float x,float y)
            {
                  part1=x;  part2=y;
            }
            void put_marks(void)
{cout << "Marks obtained:" << "\n"
       << "Part1= " <<part1<<"\n"
```

```cpp
                      <<"Part2= "<<part2<<"\n";
}
};
class sports
{        protected:
                    float score;
         public:
         void get_score(float s)
         {
                 score=s;
         }
         void put_score(void)
         {
                 cout << "Sports wt:" <<score<<"\n\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();
       put_score();
       cout<<"Total score: "<<total<< "\n";
}
int main()
{
       result stud;
       stud.get_no(1223);
       stud.get_marks(27.5, 33.0);
```

     stud.get_score(6.0);

     stud.display();

     return 0;

}

OUTPUT

Roll no 123

Marks obtained : part1=27.5

Part2=33

Sports=6

Total score = 66.5

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 substitute with required data type. Since a template defined with a parameter that would be replaced by a specified data type at the time of actual use of the class or function, the templates are sometimes called parameterized classes or functions.

Class templates

- A simple process to create a generic class using a template with anonymous type.

- template is the keyword used to create Template

- The class template definition is very similar to an ordinary class definition expect the prefix template<class T> and the use of type T.

- This prefix tells the compiler that is going to declare a template and use T as a type name in the declaration.

Syntax:

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

Example:

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

Class Templates with Multiple Parameters:

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

Syntax:

```
template <class T1, class T2,…,class Tn>
class class-name
{//body of the class
};
```

Program:

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

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:

```cpp
#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:

```cpp
#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; }
```

POSSIBLE QUESTIONS – UNIT V

Part-A

Online Examinations   (One marks)

1. Variables are declared in_____
   a) Only in main ()   **b) anywhere in the scope**
   c) After the main () function   d) the function.

2. _____statement is used to transfer the control t pass on the beginning of the block/loop
    a) Break   b) jump   c) goto   **d) continue**

3. Which of the following function/types of function cannot have default parameters?
    a) Member function of class   **b) main()**
    c) Member function of structure   d) Both B and C

4. Which is more effective while calling the functions?
    a) call by value   **b) call by reference**   c) call by pointer   d) none

5. The operator used for dereferencing or indirection is ____
    a) *   b) &   c) ->   d) ->>

6. Which is more memory efficient?
    a) Structure   **b) union**   c) both use same memory d) depends on a programmer

7. Identify the correct sentence regarding inequality between reference and pointer.

    a) We cannot create the array of reference.

    b) We can create the Array of reference.

    c) We can use reference to reference.

    d) none of the mentioned

8. The output formats can be controlled with manipulators having the header file as
    a) iostream.h   b) conio.h   c) stdlib.h   **d) iomanip.h**

9. A _____ is a collection of related data stored in a particular area on a disk.
    a) Field   b)**File**   **c)** Row   d)Vector

10. Which header file should be included to use functions like malloc() and calloc()?
    a) memory.h   **b) stdlib.h**   c) string.h   d) dos.h

Part-B

2 MARKS

1. Define constructors with example.

2. What is copy constructor?

3. Define class.

4. What is OOP?

5. Define function overloading.

6. What is operator overloading?

7. Define inheritance

8. What is polymorphism?

9. What is data abstraction and encapsulation?

10. What is an exception mention it syntax.

11. Define virtual functions.

Part-C

6 MARKS

1. Explain copy constructors with suitable example program

2. Discuss the concept of virtual function with an example program

3. What do you mean by Constructor? Give example program for constructor overloading.

4. List the different types of inheritance. Explain multi-level with suitable program.

5. Explain class constructors with suitable example program

6. List the different types of inheritance. Explain multiple with suitable program.

7. Explain the concept of function overloading with a program to find the area of triangle

8. Write note on basic exception handling in c++.

9. Explain in detail about overloading operators with example

10. Write note on catching all exceptions in c++.

Part -A Online Examinations   **(1 mark questions)**
SUBJECT: PROGRAMMING FUNDAMENTALS   SUBJECT CODE: 17CSU101
USING C/C++

UNIT-V

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	calling function	declaration	objects	none of the above	declaration
4	The class _____ describes how the class function are implemented	Function definition	declaration	arguments	none of the above	Function definition
5	The keywords private and public are known as _____ labels	Static	dynamic	visibility	const	visibility
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 declared inside the class are called as _____	Function variables	data members	member function	data variables	data members
9	The symbol _____ is called the scope resolution operator	>>	::	<<	::*	::
10	_____ enables an object to initialize itself when it is created	Destructor	constructor	overloading	none of the above	constructor
11	The _____ is special because its name is the same as the class name.	Destructor	static	constructor	none of the above	constructor
12	A constructor that accepts no parameters is called the _____ constructor	Copy	default	multiple	none of the above	default
13	Constructors are invoked automatically when the _____ are created	Datas	classes	objects	none of the above	objects
14	Constructors cannot be _____	Inherited	destroyed	both a & b	none of the above	Inherited
15	Constructors cannot be _____	Destroyed	virtual	both a & b	none of the above	virtual
16	Constructors make _____ calls to the operators new and delete when memory allocation is required	Explicit	implicit	function	none of the above	implicit
17	The constructors that can take arguments are called _____ constructors	Copy	multiple	parameterized	none of the above	parameterized

					none of the above	
18	The constructor function can also be defined as _____ function	Friend	inline	default	none of the above	inline
19	When a constructor can accept a reference to its own class as a parameter, in such cases it is called as	Multiple	copy	default	none of the above	copy
20	A _____ constructor is used to declare and initialize an object from another object	Default	copy	multiple	parameterized	copy
21	The process of initializing through a copy constructor is known as _____ initialization	Overloaded	multiple	copy	none of the above	copy
22	_____ is used to free the memory	new	delete	clrscr()	none of the above	delete
23	Which is a valid method for accessing the first element of the array item?	item(1)	item[1]	item[0]	item(0)	item[0]
24	An object is an _____ unit	group	individual	both a&b	none of the above	individual
25	Public keyword is terminated by a _____	Semicolon	comma	dot	colon	colon
26	Private keyword is terminated by a _____	semicolon	comma	dot	colon	colon
27	The memory for static data is allocated only _____	twice	thrice	once	none of the above	once
28	Static member functions can be invoked using _____ name	class	object	data	function	class

29	When a class is declared inside a function they are called as _____ classes.	global	invalid	local	none of the above	local
30	Constructors and destructors are automatically inkoved by _____	operating system	main()	complier	object	complier
31	Constructors is executed when _____	object is destroyed	object is declared	both a & b	none of the above	object is declared
32	The destructor is executed when _____	object goes out of scope	when object is not used	when object contains nothing	none of the above	object goes out of scope
33	The members of a class are by default _____	protected	private	public	none of the above	2
34	The _____ is executed at the end of the function when objects are of no used or goes out of scope	destructor	constructor	inheritance	none of the above	destructor
35	The statement catches the exception _____ .	catch	try	template	throw.	catch
36	In a multiple catch statement the number of throw statements are .	same as catch statement	twice than catch	only one	none.	only one
37	The exception is generated in _____ block.	try	catch	finally	throw.	try
38	The exception handling one of the function is implicitly invoked.	abort	exit	assert	none.	abort
39	The exception handling mechanism is basically built upon _____ keyword	try	catch	throw	all the above	all the above

40	The point at which the throw is executed is called _____.	try	catch	throw point	exceptions	throw point
41	A template function may be overloaded by _____ function	template	normal	stream	exception	template
42	_____function returns true when an input or output operation has failed	eof()	fail()	bad()	good()	fail()
43	.In _____ inheritance, the base classes are constructed in the order in which they appear in the declaration of the derived class.	Hybrid	Multipath	Hierarchical	Multiple	Multiple
44	The _____ function takes no operator.	Operator +()	Operator −()	Friend	Conversion	operator -( )
45	In overloading of binary operators, the _____ operand is used to invoke the operator function.	Right-hand	Arithmetic	Left-hand	Multiplication	left-hand
46	_____ functions may be used in place of member functions for overloading a binary operator	Inline	Member	Conversion	Friend	Friend
47	The operator that cannot be overloaded is _____	Sizee of	+	-	=	single of
48	The friend functions cannot be used to overload the _____ operator.	::	?:	.	=	::
49	_____ is called compile time polymorphism.	Operator overloading	Function overloading	Overloading unary	Overloading binary	operator overloading
50	_____ feature can be used to add two user-defined operator data types.	Function	Overloading	Arrays	Pointers	overloading

51	_____ operator cannot be overloaded.	=	+	?:	–	?:
52	Operator overloading is done with the help of a special function called _____ function.	Conversion	Operator	User-defined	In-built.	operator
53	_____ functions must either be member functions or friend functions.	Operator	User-defined	Static Member	Overloading	operator
54	The overloading operator must have atleast _____ operand that is of user-defined data type.	Two	Three	One	Four	one
55	_____ operator function should be a class member.	Arithmetic	Relational	Casting	Overloading	casting
56	The casting operator must not have any _____	Arguments	Member	Return type	Operator	arguments
57	The casting operator function must not specify a _____ type.	User-defined type	Return	Member	In-built	return
58	The operator that cannot be overloaded is _____.	Casting	Binary	Unary	Scope resolution	scope resolution
59	The friend function cannot be used to overload _____ operator.	+	-	( )	::	( )
60	_____ operator cannot be overloaded by friend function.	[]	*	.	?:	?:
61	The operator that cannot be overloaded by friend function is _____	.	::	->	Single of	::

62	Operator overloading is called _____	Function Overloading	Compile time polymorphi	Casting operator function	Temporary object	Compile time polymorphism
63	Overloading feature can add two _____ data types.	In-built	Enumerated	User-defined	Static	User-defined
64	The mechanism of deriving a new class from an old one is called _____	Operator overloading	Inheritance	Polymorphism	Access mechanism	polymorphism

Reg. No.................................

[12CSU201/12ITU201/12CAU201]

KARPAGAM UNIVERSITY

(Under Section 3 of UGC Act 1956)
COIMBATORE – 641 021
(For the candidates admitted from 2012 onwards)

B.Sc., & BCA DEGREE EXAMINATION, APRIL 2013
Second Semester

COMPUTER SCIENCE / INFORMATION TECHNOLOGY / COMPUTER APPLICATIONS

OBJECT ORIENTED PROGRAMMING WITH C++

Time: 3 hours   Maximum : 100 marks

PART – A (15 x 2 = 30 Marks)
Answer ALL the Questions

1. What are Tokens?
2. Define Variable.
3. What is mean by Inline Function?
4. Explain the keyword Static.
5. Define Class.
6. What is meant by Friend Function?
7. Define Single Inheritance.
8. Define Operator overloading.
9. Define Function Overloading.
10. Define pointer.
11. Explain Virtual function.
12. List out 2 Manipulators.
13. Define File.
14. What is the use of eof () function?
15. What is meant by Input Streams?

PART B (5 X 14= 70 Marks)
Answer ALL the Questions

16. a. Explain the types of Operators.
             Or
    b. Explain the object oriented programming language Concepts.

17. a. Explain the types of constructor with an example.
             Or
    b. Define class and object and explain the structure of class and object.

18. a. Explain the various types of Inheritance with an example.
             Or
    b. Explain the method overloading concepts with an example.

19. a. Discuss about formatted console I/O operations.
             Or
    b. Explain the concept of Manipulators with an example.

20. a. Explain command line argument in c++.
             Or
    b. Write a program for how to create a file in C++.

Reg. No.................................

Time: 3 hours   Maximum : 60 marks

PART – A (10 x 2 = 20 Marks)
Answer any TEN Questions

1. Define polymorphism.
2. Give the difference between else..if ladder and switch statements.
3. What is inline function? Give its syntax?
4. Write the syntax and example for creating an object in c++.
5. What is a friend function?
6. Give the syntax for writing a destructor in c++.
7. List out the operators that cannot be overloaded by friend functions?
8. Which operator is used for defining derived class?
9. Define hybrid inheritance.
10. What do mean pointers to objects?
11. Differentiate between input and output stream.
12. Give the general form for defining our own manipulators.
13. What is the use of open() function?
14. What are the tasks performed in the updating of a file?
15. How can a template function be overloaded?

PART B (5 X 8 = 40 Marks)
Answer ALL the Questions

16. a) Discuss in detail about the benefits of oops concept.

Or

b) Elucidate the principle of function overloading with appropriate illustrations.

17. a) Discuss in detail about static data members and static member functions with example

Or

b) Describe how a class can have multiple constructors with suitable examples.

18. a) Explain with an example how to create a pointer to derived classes with examples

Or

b) Discuss in detail about managing I/O operations with manipulators.

19. a) Illustrate how to determine number of objects in a file with an example

Or

b) Distinguish between overloaded functions and function templates in detail.

20. **Compulsory** : -

What is the procedure to overload a unary operator? Write an example program to overload increment operator to increment a date.

KARPAGAM UNIVERSITY

(Under Section 3 of UGC Act 1956)
COIMBATORE – 641 021
(For the candidates admitted from 2014 onwards)

B.Sc., DEGREE EXAMINATION, APRIL 2015

Second Semester

COMPUTER SCIENCE

OBJECT ORIENTED PROGRAMMING WITH C++

Time: 3 hours   Maximum : 60 marks

PART – A (10 x 2 = 20 Marks)
Answer any TEN Questions

1. Define polymorphism.
2. Give the difference between else..if ladder and switch statements.
3. What is inline function? Give its syntax?
4. Write the syntax and example for creating an object in c++.
5. What is a friend function?
6. Give the syntax for writing a destructor in c++.
7. List out the operators that cannot be overloaded by friend functions?
8. Which operator is used for defining derived class?
9. Define hybrid inheritance.
10. What do mean pointers to objects?
11. Differentiate between input and output stream.
12. Give the general form for defining our own manipulators.
13. What is the use of open() function?
14. What are the tasks performed in the updating of a file?
15. How can a template function be overloaded?

PART   B (5 X 8 = 40 Marks)
Answer ALL the Questions

16. a) Discuss in detail about the benefits of oops concept.
                              Or
    b) Elucidate the principle of function overloading with appropriate illustrations.

17. a) Discuss in detail about static data members and static member functions with example
                              Or
    b) Describe how a class can have multiple constructors with suitable examples.

18. a) Explain with an example how to create a pointer to derived classes with examples
                              Or
    b) Discuss in detail about managing I/O operations with manipulators.

19. a) Illustrate how to determine number of objects in a file with an example
                              Or
    b) Distinguish between overloaded functions and function templates in detail.

20. **Compulsory** : -

    What is the procedure to overload a unary operator? Write an example program to overload increment operator to increment a date.

Reg. No.....................................

[16CAU101/16CSU101]

KARPAGAM UNIVERSITY
Karpagam Academy of Higher Education
(Established Under Section 3 of UGC Act 1956)
COIMBATORE – 641 021
(For the candidates admitted from 2016 onwards)

BCA., / B.Sc., DEGREE EXAMINATION, NOVEMBER 2016
First Semester

COMPUTER APPLICATIONS/COMPUTER SCIENCE

PROGRAMMING FUNDAMENTALS USING C/C++
Time: 3 hours   Maximum : 60 marks

PART – A (20 x 1 = 20 Marks) (30 Minutes)
(Question Nos. 1 to 20 Online Examinations)

PART B (5 x 2 = 10 Marks) (2 ½ Hours)
Answer ALL the Questions

21. What is a procedural programming language?
22. What is an array?
23. Define array of structures.
24. What do you mean by preprocessor?
25. Write the Overloading assignment operator.

PART C (5 x 6 = 30 Marks)
Answer ALL the Questions

26. a. Explain the arithmetic, logical and bitwise operators with example.

Or

b. Write a program to implement Average of all Even Numbers upto a Range.

27. a. How to defining and initializing two dimensional array.

Or

b. Explain the command-line arguments.

28. a. Write the following: (i) Pointers to Pointers (ii) Pointers to Structures.

Or

b. How will you passing and returning structures from functions.

29. a. Describe about opening and closing a file.

Or

b. Differentiate between static and dynamic memory allocation.

30. a. Explain in detail about constructor overloading.

Or

b. What is a virtual function? Explain.

1

2