KARPAGAM ACADEMY OF HIGHER EDUCATION

(Deemed to be University)
(Established Under Section 3 of UGC Act 1956)
Eachanari (po), Coimbatore-21

DEPARTMENT OF CS, CA & IT

LECTURE PLAN

SUBJECT NAME: Programming Fundamentals Using C / C++ **SUBJECT CODE: 18ITU101**

SEMESTER: I

BATCH: 2018-2021 **CLASS: I B.Sc.IT**

STAFF: Dr.D.SHANMUGA PRIYAA

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	S1:1-3, 12-14 S2:4-7, 28
2.	1	➢ Using main() function, Compiling and Executing Simple Programs in C++.	S1:12-14 S2:28
3.	1	➢ **Data Types, Variables, Constants, Operators and Basic I/O:** Declaration, Defining and Initializing Variables, Scope of Variables	S1:30-31, 34-35 S2:42-46
4.	1	➢ Using Named Constants, Keywords, Data Types, Casting of Data Types	S1:25-30,31-34 S2:32-37
5.	1	➢ Operators, Using Comments in programs	S1:52-61 S2: 46-49
6.	1	➢ Character I/O, Formatted and Console I/O, Using Basic Header Files	S1:84-98 S2:21, 248-266
7.	1	➢ **Expressions, Conditional Statements and Iterative Statements:** Simple Expressions in C++ , Understanding Operators Precedence in Expressions	S1: 63-67 S2: 54-56
8.	1	➢ Conditional Statements, Understanding syntax and utility of Iterative Statements, Use of break and continue in Loops, Using Nested Statements,	S1:114-126
9.	1	Recapitulation and Discussion of important questions	S1:121-126, 152-166
	1	**Total No. of Periods allotted for Unit – I**	**9**

UNIT-II			
1.	1	Functions and Arrays: Utility of functions, Call by Value, Call by Reference	S1:270-272
2.	1	Functions returning value, Void functions, Inline Functions	S1: 269-272, 274
3.	1	Return data type of functions, Functions parameters, Differentiating between Declaration and Definition of Functions	S1:272-274
4.	1	Command Line Arguments/Parameters in Functions	S1:405-408 S2:301-303
5.	1	Functions with variable number of Arguments.	S1:285-286
6.	1	Creating and Using One Dimensional Arrays	S1: 192-199
7.	1	Various types of arrays	S1: 209-210
8.	1	Two-dimensional Arrays, Introduction to Multi-dimensional arrays.	S1:199-209
9.	1	Recapitulation and Discussion of important questions	
		Total No. of Hours allotted for Unit – II	**9**
UNIT-III			
1.	1	**Derived Data Types (Structures and Unions):** Understanding utility of structures and unions, Declaring, initializing and using simple structures and unions	S1: 371-319, 322-324
2.	1	Manipulating individual members of structures and unions	S1: 321-322
3.	1	Array of Structures, Individual data members as structures	S1: 326-329
4.	1	Passing and returning structures from functions, Structure with union as members, Union with structures as members	S1: 333-337
5.	1	**Pointers and References in C++:** Understanding a Pointer Variable, Simple use of Pointers	S1:351-355
6.	1	Pointers to Pointers, Pointers to structures Problem with Pointers	S1: 376-379
7.	1	Passing pointers as function arguments, Returning a pointer from a function	S1:370-373
8.	1	Using arrays as pointers, Passing arrays to functions.	S1:369-370
9.	1	Pointers vs. References, Declaring and initializing references, using references as function arguments and function return values	S1:371-380
10.	1	Recapitulation and Discussion of important questions	
		Total No. of Hours allotted for Unit – III	**10**

UNIT-IV			
1.	1	**Memory Allocation in C++:** Differentiating between static and dynamic memory allocation, use of malloc, calloc and free functions,	S5:469-470
2.	1	Use of new and delete operators, storage of variables in static and dynamic memory allocation.	S4:456-458
3.	1	**File I/O, Preprocessor Directives:** Opening and closing a file	S1:389-392
		Reading and writing Text Files	S1:392-394
4.	1	Using put(), get()	S1:394-398
	1	read() functions, write()functions	S1:289-294
5.	1	Random access in files	S1:400-405 S2:294-299
6.	1	Understanding the Preprocessor Directives	S2:444-445, 449-453
7.	1	Macros.	S2:445-449
8.	1	Recapitulation and Discussion of important questions	
		Total No. of Hours allotted for Unit – IV	**8**
UNIT-V			
1.	1	Using Classes in C++: Principles of Object- Oriented Programming, Defining & Using Classes	S2: 88-96
2.	1	Constructors, Constructor Overloading	S2:127-133
3.	1	Function overloading, Class Variables &Functions, Access specifiers	S2:80-82,98 107-109
4.	1	Copy Constructors,Overview of Template classes and their use	S2:308-314
5.	1	Operator overloading: Need of Overloading functions and operators, Overloading functions by number and type of arguments,	S2:150-157
6.	1	Looking at an operator as a function call, Overloading Operators	
7.	1	Inheritance, Polymorphism and Exception Handling: Introduction to Inheritance	S2:176-179
8.	1	Polymorphism	S2: 222-223
9.	1	Basics Exceptional Handling	S2:326-332
10.	1	Recapitulation and Discussion of important questions	
11.	1	Discussion of previous ESE Question papers	
12.	1	Discussion of previous ESE Question papers	
		Discussion of previous ESE Question papers	
		Total No. of Hours allotted for Unit – V	**12**

Total No. of Hours: 48

Suggested Readings

S1. BjarneStroustroup, 2014. "Programming -- Principles and Practice using C++", 2nd Edition, Addison-Wesley.

S2. BjarneStroustrup, 2013. "The C++ Programming Language", 4th Edition, Addison-Wesley.

S3. Harry, H. Chaudhary, 2014. "Head First C++ Programming: The Definitive Beginner's Guide", First Create space Inc, O-D Publishing, LLC USA.

S4. Stanley B. Lippman, JoseeLajoie, Barbara E. Moo, 2012. "C++ Primer", Published by Addison-Wesley, 5th Edition.

S5. Paul Deitel, Harvey Deitel, 2011. "C++ How to Program", 8th Edition, Prentice Hall.

S6. E Balaguruswamy, 2008. "Object Oriented Programming with C++", Tata McGraw-Hill Education.

S7. Walter Savitch, 2007. "Problem Solving with C++", Pearson Education.

S8. HerbtzSchildt.(2003). C++: The Complete Reference (4thed.)New Delhi: McGraw Hill.

S9. John, R. Hubbard. (2000). Programming with C++-(2nd ed.).Schaum's Series.

S10. Andrew Koeni., Barbara, E. Moo. (2000). Accelerated C++.Addison-Wesley.

S11. Scott Meyers. (2005). Effective C++ (3rd ed.).Addison-Wesley,.

S12. Harry, H. Chaudhary. (2014). Head First C++ Programming: The Definitive Beginner's Guide. LLC USA:First Create space Inc, O-D Publishing,.

Websites

W1. www.cplusplus.com/doc/tutorial/

W2. www.cplusplus.com/

W3. www.cppreference.com/

W4. http://www.cs.cf.ac.uk/Dave/C/CE.html

W5. http://www2.its.strath.ac.uk/courses/c/

W6. http://www.iu.hio.no/~mark/CTutorial/CTutorial.html

KARPAGAM ACADEMY OF HIGHER EDUCATION

(Deemed to be University)
(Established Under Section 3 of UGC Act 1956)
Eachanari (po), Coimbatore-21

Semester – I

18ITU101 PROGRAMMING FUNDAMENTALS USING C / C++ 4H – 4C

Instruction Hours / week: L: 4 T: 0 P: 0 **Marks:** Int : **40** Ext : **60** Total: **100**

SCOPE

C is designed for developing system software, portable application software. Despite its low-level capabilities, the language was designed to encourage cross-platform programming. Also this course introduces the concepts of Object Oriented Programming language.

COURSE OBJECTIVES

- To impart adequate knowledge on the need of programming languages and problem solving techniques.
- To develop programming skills using the fundamentals and basics of C Language.
- To enable effective usage of arrays, structures, functions, pointers and to implement the memory management concepts.
- To teach the issues in file organization and the usage of file systems.
- To learn the characteristics of an object-oriented programming language: data abstraction and information hiding, inheritance, and dynamic binding of the messages to the methods.

COURSE OUTCOMES

After the completion of this course, a successful student will be able to do the following:

- Obtain the knowledge about the number systems this will be very useful for bitwise operations.
- Develop programs using the basic elements like control statements, Arrays and Strings .
- Solve the memory access problems by using pointers
- understand about the dynamic memory allocation using pointers which is essential for utilizing memory
- Understand about the code reusability with the help of user defined functions.
- Develop advanced applications using enumerated data types, function pointers and nested structures.
- Learn the basics of file handling mechanism that is essential for understanding the concepts in database management systems.
- Understand the uses of preprocessors and various header file directives.
- Use the characteristics of an object-oriented programming language in a program.
- Use the basic object-oriented design principles in computer problem solving.

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 ed.) New Delhi: McGraw Hill.
2. Bjarne Stroustrup. (2013). The C++ Programming Language(4th ed.). New Delhi: Addison-Wesley.
3. Bjarne Stroustroup. (2014). Programming, Principles and Practice using C++(2nd ed.). New Delhi: Addison-Wesley.
4. Balaguruswamy, E. (2008). Object Oriented Programming with C++. New Delhi: Tata McGraw-Hill Education.
5. Paul Deitel., & Harvey Deitel. (2011). C++ How to Program (8th ed.). New Delhi: Prentice Hall.
6. John, R. Hubbard. (2000). Programming with C++- (2nd ed.). Schaum's Series.
7. Andrew Koeni., Barbara, E. Moo. (2000). Accelerated C++. Addison-Wesley.
8. Scott Meyers. (2005). Effective C++ (3rd ed.).Addison-Wesley,.
9. Harry, H. Chaudhary. (2014). Head First C++ Programming: The Definitive Beginner's Guide. LLC USA: First Create space Inc, O-D Publishing,.
10. Walter Savitch.(2007) Problem Solving with C++, Pearson Education,.
11. Stanley, B. Lippman., Josee Lajoie., & Barbara, E. Moo. (2012). C++ Primer, 5th ed.). 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. www.cplusplus.com/
6. www.cppreference.com/

ESE Pattern	
Part – A (Online)	20 * 1 = 20
Part – B	5 * 2 = 10
Part – C (Either or)	5 * 6 = 30
Total	60 marks

Faculty **HOD**

UNIT-I

Syllabus

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)

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.
1. 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

1 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 :
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.

History of C Language:
In 1960's COBOL was being used for commercial applications and FORTRAN is used for scientific and engineering applications. At this stage people started to develop a language which is suitable for all possible applications. Therefore an international committee was setup to develop such a language " ALGOL 60 " was released. It was not popular , because it seemed too general. To reduce these generality a new language CPL(combined programming language) was developed at Cambridge University. It has very less features. Then some other features were added to this language and a new language called BCPL(Basic combined programming language) developed by "Martin Richards" at Cambridge University. Then " B " language was developed by "Ken Thompson" at AT&T BELL labs. Dennis Ritchie inherited the features of B and BCPL, added his own features and developed C language in 1972.

Features of 'C' Language:
1. C is a structured programming language with fundamental flow control construction.
2 . C is simple and versatile language.
3. Programs written in C are efficient and fast.

4. C has only 32 keywords.
5. C is highly portable programming language. The programs written for one computer can be run on another with or without any modifications
6. C has rich set of operators.
7. C permits all data conversions and mixed mode operations
8. Dynamic memory allocation(DMA) is possible in C.
9. Extensive varieties of data types such as arrays, pointers, structures and unions are available in C.
10. C improves by itself. It has several predefine functions.
11. C easily manipulates bits, bytes and addresses.
12. Recursive function calls for algorithmic approach is possible in C.
13. Mainly we are using C language to implement system softwares. These are compilers ,editors, drivers ,databases and operating systems.
14. C compiler combines the capability of an assembly level language with the features of high level language. So it is called as middle level language.

Once the program is completed, the program is feed into the computer using the compiler to produce equivalent machine language code. In C program compilation there are 2 mechanisms.

1. Compiler
2. Linker.

The Compiler receives the source file as input and converts that file into object file. Then the **Linker** receives the object file as its input and linking with C libraries. After linking it produces an executable file for the given code. After creation of executable file, then start the program execution and loads the information of the program into primary memory through LOADER. After loading the information the processor processing the information and gives output.

Compiler	Interpreter
Compiler Takes Entire program as input	Interpreter Takes Single instruction as input
Intermediate Object Code is Generated	No Intermediate Object Code is Generated
Conditional Control Statements are Executes faster	Conditional Control Statements are Executes slower
Memory Requirement : More (Since Object Code is Generated)	Memory Requirement is Less
Program need not be compiled every time	Every time higher level program is converted into lower level program
Errors are displayed after entire program is checked	Errors are displayed for every instruction interpreted (if any)
Example : C Compiler	Example : BASIC

Basic structure of C program

[Document section]
Preprocessor section
(or)
Link section
[Global declaration section]
main( )

```
{
    [ Local declaration section ]
     Statements
}
[Sub program section]
(include user defined functions)
```

Document section:

It consists of a set of comment lines giving the name of the program, author name and some other details about the entire program.

Preprocessor Section (or) Link section:

It provides instructions to the compiler to link the functions from the system library.

Global declaration Section:

The Variables that are used in more than one function are called as global variables and these are declared in global declaration section.

Main function section:

Every C program must have one function. i.e. main function. This section contains 2 parts.

1. Local declaration section.
1. Statements .

The Local declaration section declares all the variables used in statements. The statements part consists a sequence of executable statements.These 2 parts must appear between opening and closing curly braces. The program execution begins at the opening brace and ends at closing brace.

Sub programming section:

It contains all the user defined functions.
This section may be placed before or after main function.

Comments:

Unexecutable lines in a program are called as comments. These lines are skipped by the compiler.

```
/*----------------------
  ----------------------*/   Multi line comment.
//----------   single line comment(C++ comment).
```

Preprocessor statements:

The preprocessor is a program that process the source code before it passes through the compiler.

#include:

It is preprocessor file inclusion directive and is used to include header files. It provides instructions to the compiler to link the functions from the system library.

Syntax:

```
#include   " file_name "
        (or)
#include   < file_name >
```

When the file name is included within the double quotation marks, the search for the file is made first in the current directory and then the standard directories(TC). Otherwise when the file name is included within angular braces, the file is search only in standard diretories(TC).

stdio.h :- standard input and output header file.
conio.h :- console input and output header file.
console units: keyboard and monitor.

These 2 headers are commonly included in all C programs.

If you want to work with Turbo c/c++ ,first install Turbo c/c++ software and then follow the following steps.

Steps involved in C programming:

1) How to open a C editor(windows xp):
   1. Start Menu □ Run □ type C:\TC\Bin\TC.exe then press Enter key
   2. At the time of installation create Shortcut to C on Desktop, then it will create an icon(TC) on desktop. Double click on that icon.

How to open a C editor(windows 7):
   1. At the time of installation it creates Shortcut to C (turboc++)on Desktop. Double click on that icon.
   2. After entering into C editor, check the path as:
      goto Options menu -> Directories □
□□□□windows xp
      1. C :\TC\Include
      2. C :\TC\Lib

3) Type C program , goto File menu □ New
4) Save program , goto File menu □ Save
5) Compile Program ,
   goto Compile menu □ Compile
6) Run the Program , goto Run menu □ Run
7) See the output ,
   goto window menu -> user screen
8) Exit from C editor , goto File menu □ Quit

Shortcut Keys:

Open	:	F3
Save	:	F2
Close file	:	Alt + F3
Full Screen	:	F5
Compile	:	Alt + F9
Run	:	Ctrl + F9
Output	:	Alt + F5
Change to another file:	:	F6
Help	:	Ctrl + F1
Tracing	:	F7
Quit	:	Alt + X

CHARACTER SET

C characters are grouped into the following categories.
1. Letters
2. Digits
3. Special Characters
4. White Spaces

Note: The compiler ignores white spaces unless they are a part of a string constant

Letters
Uppercase A….Z
Lowercase a…..z
Digits
All decimal digits 0…..9

Special characters

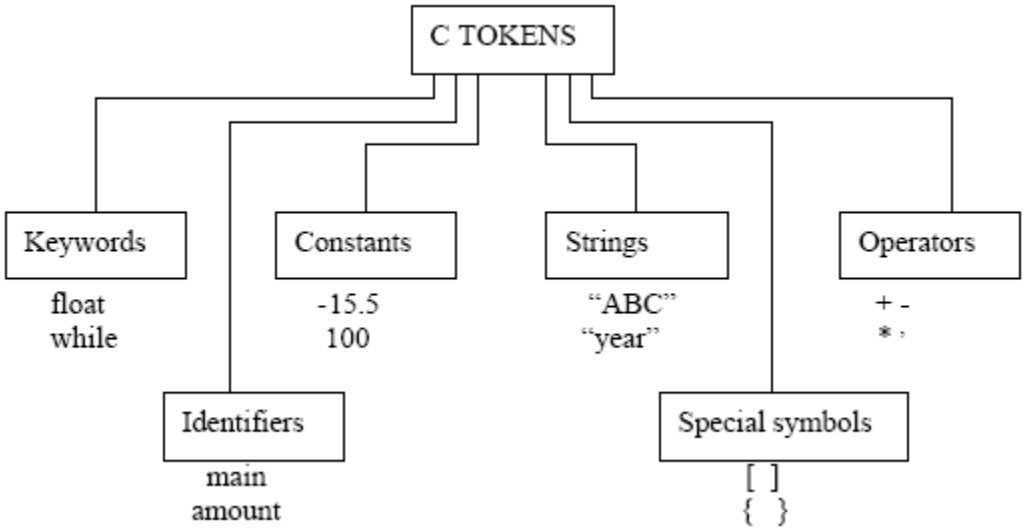
,	Comma	&	Ampersand
.	Period	^	Caret
;	Semicolon	*	Asterisk
:	Colon	-	Minus
?	Question mark	+	Plus sign
'	Apostrophe	<	Less than
"	Quotation mark	>	Greater than
!	Exclamation	(	Left parenthesis
|	Vertical Bar	)	Right parentheses
/	Slash	[	Left bracket
\	Back slash	]	Right bracket
~	Tilde	{	Left brace
_	Underscore	}	Right brace
$	Dollar sign	#	Number sign
%	Percent sign		

White Spaces

- Blank Space
- Horizontal Tab
- Carriage Return
- New Line
- Form Feed

C tokens

In C programs, the smallest individual units are known as tokens.


C TOKENS
Keywords
float
while
Constants
-15.5
100
Strings
"ABC"
"year"
Operators
+ -
* ,
Identifiers
main
amount
Special symbols
[ ]
{ }

KAHE

Keywords and Identifiers

Every C word is classified as either a keyword or an identifier. All keywords have fixed meanings and these meanings cannot be changed.

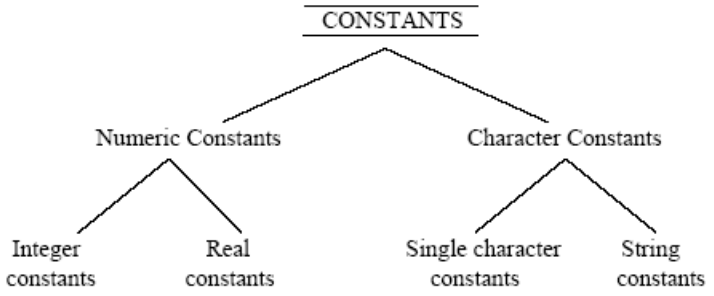
Eg: auto, break, char, void etc.,

Identifiers refer to the names of variables, functions and arrays. They are user-defined names and consist of a sequence of letters and digits, with a letter as a first character. Both uppercase and lowercase letters are permitted. The underscore character is also permitted in identifiers.

auto	**double**	**int**	**struct**
break	**else**	**long**	**switch**
case	**enum**	**register**	**typedef**
char	**extern**	**return**	**union**
const	**float**	**short**	**unsigned**
continue	**for**	**signed**	**void**
default	**goto**	**sizeof**	**volatile**
do	**if**	**static**	**while**

Constants

Constants in C refer to fixed values that do not change during the execution of a program.



Integer Constants

An integer constant refers to a sequence of digits, There are three types integers, namely, decimal, octal, and hexa decimal.

Decimal Constant

Eg:123,-321 etc.,

Note: Embedded spaces, commas and non-digit characters are **not** permitted between digits.

Eg: 1) 15 750 2)$1000

Octal Constant

An octal integer constant consists of any combination of digits from the set 0 through 7, with a leading 0.

Eg: 1) 037 2) 0435

Hexadecimal Constant

A sequence of digits preceded by 0x or 0X is considered as hexadecimal integer. They may also include alphabets A through F or a through f.

Eg: 1) 0X2 2) 0x9F 3) 0Xbcd

Real Constants

Certain quantities that vary continuously, such as distances, heights etc., are represented by numbers containing functional parts like 17.548.Such numbers are called real (or floating point)constants.
Eg:0.0083,-0.75 etc.,
A real number may also be expressed in exponential or scientific notation.
Eg:215.65 may be written as 2.1565e2

Single Character Constants

A single character constants contains a single character enclosed within a pair of single quote marks.
Eg: '5'
'X'
'.'

String Constants

A string constant is a sequence of characters enclosed in double quotes. The characters may be letters, numbers, special characters and blank space.
Eg:"Hello!"
"1987"
"?….!"

Backslash Character Constants

C supports special backslash character constants that are used in output functions. These character combinations are known as escape sequences.

Constant	Meaning
'\a'	audible alert
'\b'	backspace
'\f'	form feed
'\n'	new line
'\0'	null
'\v'	vertical tab
'\t'	horizontal tab
'\r'	carriage return

Variables

Definition:

A variable is a data name that may be used to store a data value. A variable may take different values at different times of execution and may be chosen by the programmer in a meaningful way. It may consist of letters, digits and underscore character.
Eg: 1) Average
2) Height

Rules for defining variables

- They must begin with a letter. Some systems permit underscore as the first character.
- ANSI standard recognizes a length of 31 characters. However, the length should not be normally more than eight characters.
- Uppercase and lowercase are significant.
- The variable name should not be a keyword.
- White space is not allowed.

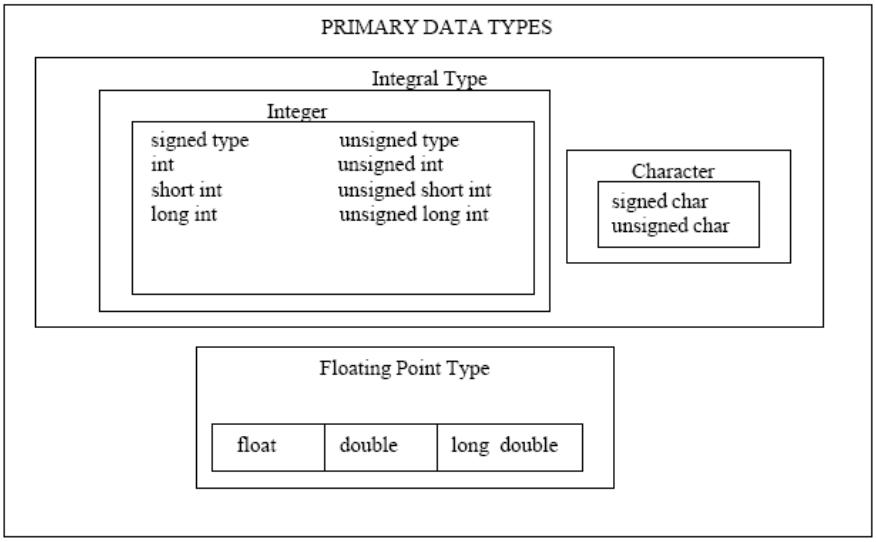
The syntax is

Data-type v1,v2…..vn;

for e.g:
int a,b;
float x,y;
Here a and b are the variables which can hold integer data. x and y are variable to hold float data. we can also assign value to the variable at the time of declaration for e.g:-
int a=26;
Here a is the name of variable and 26 is stored in this variable.

Assigning values to variables
The syntax is
Variable_name=constant
Eg:1) int a=20;
2) bal=75.84;
3) yes='x';
C permits multiple assignments in one line.
Example:
initial_value=0;final_value=100;
Declaring a variable as constant
Eg: 1) **const int** class_size=40;
This tells the compiler that the value of the int variable class_size must not be modified by the program.
Declaring a variable as volatile
By declaring a variable as volatile, its value may be changed at any time by some external source.
Eg:1) **volatile int** date;

PRIMARY DATA TYPES
The type of data which a variable can store is called its data type. C language supports following data types:-



Keyword	Description	Low	High	Bytes	Format string
Char	Single character	-128	127	1	%c
Int	Integer	-32768	32767	2	%d
Long int	Long int	-2147483648	2147483848	4	%ld
Float	Floating	3.4 e-38	3.4 e 38	4	%f
Double	Double floating	1.7 e -308	1.7 e 308	8	%lf
Long double	Long double floating	3.4 e -4932	1.1 e 4932	10	%Lf
Unsigned char	Char with no sign	0	255	1	%c
Unsign int	Int with no sign	0	65535	2	%u
Unsign long int	Long int no sign	0	4294967295	4	%lu

Operators and expressions

C programming language provides several operators to perform different kind to operations. There are operators for assignment, arithmetic functions, logical functions and many more. These operators generally work on many types of variables or constants, though some are restricted to work on certain types. Most operators are binary, meaning they take two operands. A few are unary and only take one operand

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

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Assignment Operators
- Misc Operators

Arithmetic Operators

Following table shows all the arithmetic operators supported by C language. Assume variable **A** holds 10 and variable **B** holds 20 then:

Operator	Description	Example
+	Adds two operands	A + B will give 30
-	Subtracts second operand from the first	A - B will give -10
*	Multiplies both operands	A * B will give 200
/	Divides numerator by de-numerator	B / A will give 2
%	Modulus Operator and remainder of after an integer division	B % A will give 0
++	Increments operator increases integer value by one	A++ will give 11
--	Decrements operator decreases integer value by one	A-- will give 9

Relational Operators

Following table shows all the relational operators supported by C language. Assume variable **A** holds 10 and variable **B** holds 20, then:

Operator	Description	Example
==	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A == B) is not true.
!=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

Logical Operators

Following table shows all the logical operators supported by C language. Assume variable **A** holds 1 and variable **B** holds 0, then:

Operator	Description	Example
&&	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.

Bitwise Operators

Bitwise operator works on bits and perform 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

Assume if A = 60; and B = 13; now in binary format they will be as follows:
A = 0011 1100
B = 0000 1101

A&B = 0000 1100
A|B = 0011 1101

A^B = 0011 0001
~A = 1100 0011

The Bitwise operators supported by C language are listed in the following table. Assume variable A holds 60 and variable B holds 13, then:

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12, which is 0000 1100
|	Binary OR Operator copies a bit if it exists in either operand.	(A | B) will give 61, which is 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) will give 49, which is 0011 0001
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) will give -61, which is 1100 0011 in 2's complement form.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 will give 240 which is 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15 which is 0000 1111

Assignment Operators

There are following assignment operators supported by C language:

Operator	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand	C = A + B will assign value of A + B into C
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand	C %= A is equivalent to C = C % A
<<=	Left shift AND assignment operator	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator	C &= 2 is same as C = C & 2
^=	bitwise exclusive OR and assignment operator	C ^= 2 is same as C =

		C ^ 2
|=	bitwise inclusive OR and assignment operator	C |= 2 is same as C = C | 2

Misc Operators : sizeof & ternary

There are few other important operators including **sizeof** and **? :** supported by C Language.

Operator	Description	Example
sizeof()	Returns the size of an variable.	sizeof(a), where a is integer, will return 4.
&	Returns the address of an variable.	&a; will give actual address of the variable.
*	Pointer to a variable.	*a; will pointer to a variable.
? :	Conditional Expression	If Condition is true ? Then value X : Otherwise value Y

Operators Precedence in C

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator.

For example x = 7 + 3 * 2; here, x is assigned 13, not 20 because operator * has higher precedence than +, so it first gets multiplied with 3*2 and then adds into 7.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

Operator	Description	Associativity
( )	Parentheses (function call) (see Note 1)	left-to-right
[ ]	Brackets (array subscript)	
.	Member selection via object name	
->	Member selection via pointer	
++ --	Postfix increment/decrement (see Note 2)	
++ --	Prefix increment/decrement	right-to-left
+ -	Unary plus/minus	
! ~	Logical negation/bitwise complement	
(type)	Cast (convert value to temporary value of type)	
*	Dereference	
&	Address (of operand)	
sizeof	Determine size in bytes on this implementation	
* / %	Multiplication/division/modulus	left-to-right
+ -	Addition/subtraction	left-to-right
<< >>	Bitwise shift left, Bitwise shift right	left-to-right
< <=	Relational less than/less than or equal to	left-to-right
> >=	Relational greater than/greater than or equal to	
== !=	Relational is equal to/is not equal to	left-to-right
&	Bitwise AND	left-to-right
^	Bitwise exclusive OR	left-to-right
|	Bitwise inclusive OR	left-to-right

&&	Logical AND	left-to-right
||	Logical OR	left-to-right
? :	Ternary conditional	right-to-left
=	Assignment	right-to-left
+= -=	Addition/subtraction assignment	
*= /=	Multiplication/division assignment	
%= &=	Modulus/bitwise AND assignment	
^= |=	Bitwise exclusive/inclusive OR assignment	
<<= >>=	Bitwise shift left/right assignment	
,	Comma (separate expressions)	left-to-right

Note 1:

Parentheses are also used to group sub-expressions to force a different precedence; such parenthetical expressions can be nested and are evaluated from inner to outer.

Note 2:

Postfix increment/decrement have high precedence, but the actual increment or decrement of the operand is delayed (to be accomplished sometime before the statement completes execution). So in the statement **y = x * z++;** the current value of **z** is used to evaluate the expression (i.e., **z++** evaluates to **z**) and **z** only incremented after all else is done.

I/O Functions In C

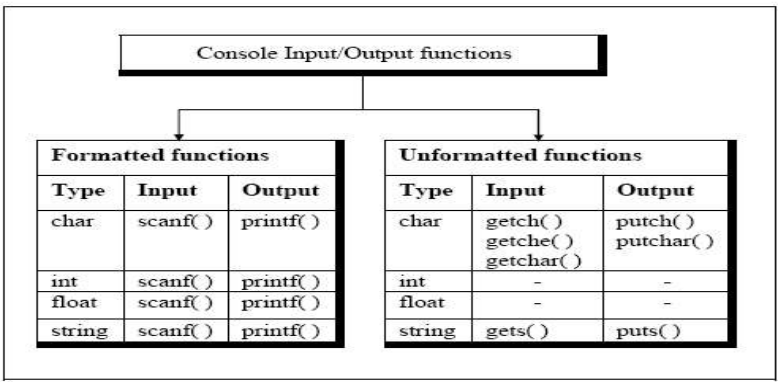
There are numerous library functions available for I/O. These can be classified into two broad categories:

(a)Console I/O functions-Functions to receive input from keyboard and write output to VDU.

(b) File I/O functions- Functions to perform I/O operations on a floppy disk or hard disk.

Console I/O Functions

The screen and keyboard together are called a console. Console I/O functions can be further classified into two categories—formatted and unformatted console I/O functions. The basic difference between them is that the formatted functions allow the input read from the keyboard or the output displayed on the VDU to be formatted as per our requirements. For example, if values of average marks and percentage marks are to be displayed on the screen, then the details like where this output would appear on the screen, how many spaces would be present between the two values, the number of places after the decimal points, etc. can be controlled using formatted functions. The functions available under each of these two categories are shown in Figure below:



Formatted Console I/O Functions

The functions **printf( ),** and **scanf( )** fall under the category of formatted console I/O functions. These functions allow us to supply the input in a fixed format and let us obtain the output in the specified form.

printf( ):

Its general form looks like this...

printf ("format string", list of variables) ;

The format string can contain:

- Characters that are simply printed as they are
- Conversion specifications that begin with a % sign
- Escape sequences that begin with a \ sign

For example, look at the following program:

```
main( )
{
int avg = 346 ;
float per = 69.2 ;
printf ( "Average = %d\nPercentage = %f", avg, per ) ;
}
```

The output of the program would be...

```
Average = 346
Percentage = 69.200000
```

How does **printf( )** function interpret the contents of the format string. For this it examines the format string from left to right. So long as it doesn't come across either a **%** or a \ it continues to dump the characters that it encounters, on to the screen. In this example **Average =** is dumped on the screen. The moment it comes across a conversion specification in the format string it picks up the first variable in the list of variables and prints its value in the specified format. In this example, the moment **%d** is met the variable **avg** is picked up and its value is printed. Similarly, when an escape sequence is met it takes the appropriate action. In this example, the moment **\n** is met it places the cursor at the beginning of the next line. This process continues till the end of format string is not reached.

Format Specifications

The **%d** and **%f** used in the **printf( )** are called format specifiers. They tell **printf( )** to print the value of **avg** as a decimal integer and the value of per as a float. Following is the list of format specifiers that can be used with the **printf( )** function.

Data type		Format specifier
Integer	short signed	%d or %I
	short unsigned	%u
	long singed	%ld
	long unsigned	%lu
	unsigned hexadecimal	%x
	unsigned octal	%o
Real	float	%f
	double	%lf
Character	signed character	%c
	unsigned character	%c
String		%s

We can provide following optional specifiers in the format specifications.

Specifier	Description
dd	Digits specifying field width
.	Decimal point separating field width from precision (precision stands for the number of places after the decimal point)
dd	Digits specifying precision
-	Minus sign for left justifying the output in the specified field width

The field-width specifier tells **printf( )** how many columns on screen should be used while printing a value. For example, **%10d** says, "print the variable as a decimal integer in a field of 10 columns". If the value to be printed happens not to fill up the entire field, the value is right justified and is padded with blanks on the left. If we include the minus sign in format specifier (as in **%-10d**), this means left justification is desired and the value will be padded with blanks on the right. Here is an example that should make this point clear.

```
main( )
{
int weight = 63 ;
printf ( "\nweight is %d kg", weight ) ;
printf ( "\nweight is %2d kg", weight ) ;
printf ( "\nweight is %4d kg", weight ) ;
printf ( "\nweight is %6d kg", weight ) ;
printf ( "\nweight is %-6d kg", weight ) ;
```

```
}
```
The output of the program would look like this ...
```
Columns 012345678901234567890123456789O
weight is 63 kg
weight is 63 kg
weight is 63 kg
weight is 63 kg
weight is 63 kg
```
Specifying the field width can be useful in creating tables of numeric values, as the following program demonstrates.
```
main( )
{
printf ( "\n%f %f %f", 5.0, 13.5, 133.9 ) ;
printf ( "\n%f %f %f", 305.0, 1200.9, 3005.3 ) ;
}
```
And here is the output...
```
5.000000 13.500000 133.900000
305.000000 1200.900000 3005.300000
```
Even though the numbers have been printed, the numbers have not been lined up properly and hence are hard to read. A better way would be something like this...
```
main( )
{
printf ( "\n%10.1f %10.1f %10.1f", 5.0, 13.5, 133.9 ) ;
printf ( "\n%10.1f %10.1f %10.1f", 305.0, 1200.9, 3005.3 );
}
```
This results into a much better output...
```
012345678901234567890123456789O1
5.0 13.5 133.9
305.0 1200.9 3005.3
```
The format specifiers could be used even while displaying a string of characters. The following program would clarify this point:
```
/* Formatting strings with printf( ) */
main( )
{
char firstname1[ ] = "Sandy" ;
char surname1[ ] = "Malya" ;
char firstname2[ ] = "AjayKumar" ;
char surname2[ ] = "Gurubaxani" ;
printf ( "\n%20s%20s", firstname1, surname1 ) ;
printf ( "\n%20s%20s", firstname2, surname2 ) ;
}
```
And here's the output...
```
012345678901234567890123456789012345678901234567890
Sandy Malya
AjayKumar Gurubaxani
```

The format specifier **%20s** reserves 20 columns for printing a string and then prints the string in these 20 columns with right justification. This helps lining up names of different lengths properly. Obviously, the format **%-20s** would have left justified the string.

Escape Sequences

We saw earlier how the newline character, **\n**, when inserted in a **printf( )**'s format string, takes the cursor to the beginning of the next line. The newline character is an 'escape sequence', so called because the backslash symbol (\) is considered as an 'escape' character—it causes an escape from the normal interpretation of a string, so that the next character is recognized as one having a special meaning.

The following example shows usage of **\n** and a new escape sequence \t, called 'tab'. A **\t** moves the cursor to the next tab stop. A 80-column screen usually has 10 tab stops. In other words, the screen is divided into 10 zones of 8 columns each. Printing a tab takes the cursor to the beginning of next printing zone. For example, if cursor is positioned in column 5, then printing a tab takes it to column 8.

```
main( )
{
printf ( "You\tmust\tbe\tcrazy\nto\thate\tthis\tbook" ) ;
}
```

And here's the output...

```
1 2 3 4
01234567890123456789012345678901234567890
You must be crazy
to hate this book
```

The **\n** character causes a new line to begin following 'crazy'. The tab and newline are probably the most commonly used escape sequences, but there are others as well. Figure shows a complete list of these escape sequences.

Esc. Seq.	Purpose	Esc. Seq.	Purpose
\n	New line	\t	Tab
\b	Backspace	\r	Carriage return
\f	Form feed	\a	Alert
\'	Single quote	\"	Double quote
\\	Backslash		

The first few of these escape sequences are more or less self- explanatory. **\b** moves the cursor one position to the left of its current position. **\r** takes the cursor to the beginning of the line in which it is currently placed. **\a** alerts the user by sounding the speaker inside the computer. Form feed advances the computer stationery attached to the printer to the top of the next page. Characters that are ordinarily used as delimiters... the single quote, double quote, and the backslash can be printed by preceding them with the backslash. Thus, the statement,

```
printf ( "He said, \"Let's do it!\"" ) ;
```

will print...

```
He said, "Let's do it!"
```

scanf( ) function:
scanf( ) allows us to enter data from keyboard that will be formatted in a certain way.
The general form of **scanf( )** statement is as follows:
scanf ("format string", list of addresses of variables) ;

For example:
scanf ("%d %f %c", &c, &a, &ch) ;

Note that we are sending addresses of variables (addresses are obtained by using '**&**' the 'address of' operator) to **scanf( )** function. This is necessary because the values received from keyboard must be dropped into variables corresponding to these addresses. The values that are supplied through the keyboard must be separated by either blank(s), tab(s), or newline(s). Do not include these escape sequences in the format string.
All the format specifications that we learnt in **printf( )** function are applicable to **scanf( )** function as well.

***sprintf( )* and *sscanf( )* Functions**
The **sprintf( )** function works similar to the **printf( )** function except for one small difference. Instead of sending the output to the screen as **printf( )** does, this function writes the output to an array of characters. The following program illustrates this.

```
main( )
{
int i = 10 ;
char ch = 'A' ;
float a = 3.14 ;
char str[20] ;
printf ( "\n%d %c %f", i, ch, a ) ;
sprintf ( str, "%d %c %f", i, ch, a ) ;
printf ( "\n%s", str ) ;
}
```

In this program the **printf( )** prints out the values of **i**, **ch** and **a** on the screen, whereas **sprintf( )** stores these values in the character array **str**. Since the string **str** is present in memory what is written into **str** using **sprintf( )** doesn't get displayed on the screen. Once **str** has been built, its contents can be displayed on the screen. In our program this was achieved by the second **printf( )** statement.
The counterpart of **sprintf( )** is the **sscanf( )** function. It allows us to read characters from a string and to convert and store them in C variables according to specified formats. The **sscanf( )** function comes in handy for in-memory conversion of characters to values. You may find it convenient to read in strings from a file and then extract values from a string by using **sscanf( )**. The usage of **sscanf( )** is same as **scanf( )**, except that the first argument is the string from which reading is to take place.

Unformatted Console I/O Functions
getchar( ), getch( ) and **getche( ) Functions:**

getch( ) and **getche( )** are two functions which return the character that has been most recently typed. The 'e' in **getche( )** function means it echoes (displays) the character that you typed to the screen. As

against this **getch( )** just returns the character that you typed without echoing it on the screen. **getchar( )** works similarly and echo's the character that you typed on the screen, but unfortunately requires Enter key to be typed following the character that you typed. The difference between **getchar( )** and **fgetchar( )** is that the former is a macro whereas the latter is a function. Here is a sample program that illustrates the use of these functions.

```
main( )
{
char ch ;
printf ( "\nPress any key to continue" ) ;
getch( ) ; /* will not echo the character */
printf ( "\nType any character" ) ;
ch = getche( ) ; /* will echo the character typed */
printf ( "\nType any character" ) ;
getchar( ) ; /* will echo character, must be followed by enter key */
printf ( "\nContinue Y/N" ) ;
fgetchar( ) ; /* will echo character, must be followed by enter key */
}
```

And here is a sample run of this program...

```
Press any key to continue
Type any character B
Type any character W
Continue Y/N Y
```

a character on the screen. As far as the working of putch( ) putchar( ) and fputchar( ) is concerned it's exactly same. The following program illustrates this.

```
main( )
{
char ch = 'A' ;
putch ( ch ) ;
putchar ( ch ) ;
fputchar ( ch ) ;
putch ( 'Z' ) ;
putchar ( 'Z' ) ;
fputchar ( 'Z' ) ;
}
```

And here is the output...

```
AAAZZZ
```

The limitation of putch( ), putchar( ) and fputchar( ) is that they can output only one character at a time.

gets( ) and puts( ) functions:

gets( ) receives a string from the keyboard. Why is it needed? Because scanf( ) function has some limitations while receiving string of characters, as the following example illustrates...

```
main( )
{
char name[50] ;
printf ( "\nEnter name " ) ;
```

```
scanf ( "%s", name ) ;
printf ( "%s", name ) ;
}
```

And here is the output...

```
Enter name Jonty Rhodes
Jonty
```

Surprised? Where did "Rhodes" go? It never got stored in the array name[ ], because the moment the blank was typed after "Jonty" scanf( ) assumed that the name being entered has ended. The result is that there is no way (at least not without a lot of trouble on the programmer's part) to enter a multi-word string into a single variable (name in this case) using scanf( ). The solution to this problem is to use gets( ) function. As said earlier, it gets a string from the keyboard. It is terminated when an Enter key is hit. Thus, spaces and tabs are perfectly acceptable as part of the input string. More exactly, gets( ) gets a newline (\n) terminated string of characters from the keyboard and replaces the \n with a \0.

The puts( ) function works exactly opposite to gets( ) function. It outputs a string to the screen.

Here is a program which illustrates the usage of these functions:

```
main( )
{
char footballer[40] ;
puts ( "Enter name" ) ;
gets ( footballer ) ; /* sends base address of array */
puts ( "Happy footballing!" ) ;
puts ( footballer ) ;
}
```

Following is the sample output:

```
Enter name
Jonty Rhodes
Happy footballing!
Jonty Rhodes
```

Why did we use two puts( ) functions to print "Happy footballing!" and "Jonty Rhodes"? Because, unlike printf( ), puts( ) can output only one string at a time. If we attempt to print two strings using puts( ), only the first one gets printed. Similarly, unlike scanf( ), gets( ) can be used to read only one string at a time.

CONTROL STATEMENTS

Normally the statements of a program will executed sequentially, one by one from the **main( )** function. Using the control statements can alter the sequence of execution. These statements also can be used to take some decisions, repeating the process number of times etc.

Any one of the following control statements will decide the order of execution.

1. Unconditional control statement
2. Conditional control statement

1. Unconditional Control Statement:

The sequence of execution can be transferred unconditionally to other part of the program using the **unconditional control statements**. The **goto** statement, is the unconditional control statement used for

this purpose using which control can be transferred to anywhere within the program without checking any condition. The format of this statement is as follows

```
goto label;
```

Where **label** is the location where the control has to be transferred. Name of the **label** is just like any identifier and is followed by a colon (**:**). Example for the label declaration in the main( ) function, is **start.**

```
main( )
{
      /* Statements;    */
  start:
      /* Statements after label to be executed; */
}
```

Example for the goto statement and label is

```
goto start;
```

Here the control is transferred to the specified label **start** and execution is continued after the label.
The control can be transferred above or below **goto** statement. If the control is transferred above, it is called **backward jump.**If the control is transferred below the **goto** statement, it is called **forward jump**. The example is given below.

```
/* Example for goto (Backward and Forward) */
main( )
{     int n,s=0;
  read1:          /* Label to read repeated values */
        printf("\nEnter the number ( 0 to Exit ) ");
        scanf("%d",&n);
        if (n==0)
              goto end;    /* Transferred to end */
        else
        {     s=s+n;
              goto read1;  /* Transferred to read1 */
        }
end:
        printf("\nSum : %d ",s);
        getch( );
}
```

Output:

```
Enter the number ( 0 to Exit ) 5
Enter the number ( 0 to Exit ) 7
Enter the number ( 0 to Exit ) 0
Sum : 12
```

2. Conditional Control statements:

The **goto**, unconditional control statement is used to transfer the control without checking any condition. This is not always preferable one. The conditional control statements are used to check some condition and then transfer the control based on the condition result. There are few control statements available and they are

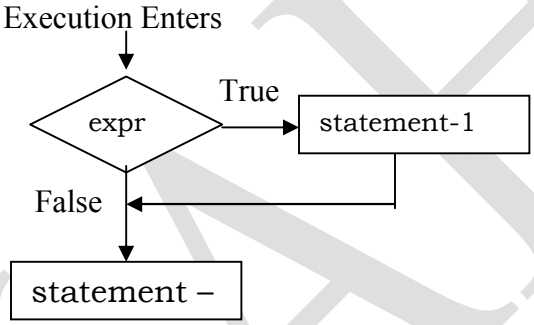
1. Simple if
2. If – else
3. Nested if
4. Switch

1. Simple if - for Small Comparison

This control statement is used to check a condition and based on the result the order of execution will be changed. The generals format and flow-chart for a simple **if** statement is as follows.

```
if ( expr)          // A Place to check the condition
    {
      statement-1;
    }
statement-n;
```

Flowchart is



Here **expr** is an expression and the result of the **expr**ession may be **TRUE / FALSE.**

If the result of expression is **TRUE,** then the **statement-1** part will be executed. Otherwise the control jumps to the **statement-n** part and continues the execution.The **statement-1** can be a simple statement or a compound statement. The compound statement must be enclosed with in braces **{ }**.

```
/* Example for simple if statement    */
        main( )
        {
           int  a;
           printf("\nEnter a number");
       scanf("%d",&a);
           if (a>0)
                printf("\nThe number is positive");
        }
```

Output:

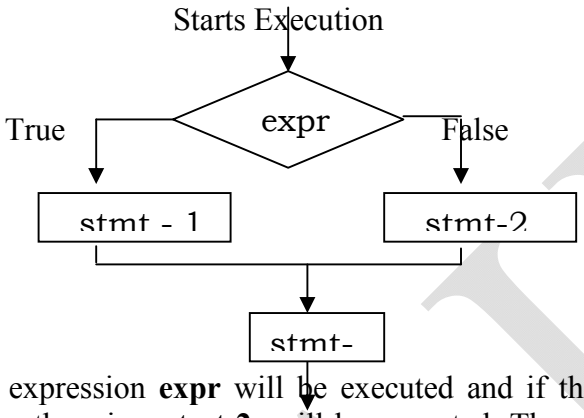
```
Enter a number  5
The number is  positive
Enter a number  -5
```

2 if - else

By using the simple **if** statement we can execute only one set of statement(s) depending upon the condition **TRUE/FALSE**. What we should do? If there are two possible results (TRUE/FALSE). We have a solution with another type of control statement **if-else.** The general format of **if-else** is illustrated with flowchart is as follows.

```
if (expr)
   stmt –1;          // TRUE Part
else
   stmt-2;           // FALSE Part
stmt-n;
```

Flow chart



First the expression **expr** will be executed and if the result is **TRUE**, then the **stmt-1** part will be executed, otherwise **stmt-2** will be executed. There is no chance to execute both statements (**stmt-1** and **stmt-2**) simultaneously. After the execution of any **stmt-1** (or) **stmt-2** process continues from the **stmt-n.** The statements may be simple or compound statements.

```
/* Example for if - else statement is here. */
main( )
{   int n;
   printf("\nEnter a number to check");
   scanf("%d", &n);

   if (n%2 = = 0)
         printf("\n%d is an even number",n);
   else
         printf("\n%d is an odd number",n);
}
```

Output:

```
Enter a number to check 5
5 is an odd number
```

The value of **a** and **b** are compared using the relational operator >. When result of **a>b** is **TRUE,** value of **a** will be assigned to the variable **big,** otherwise the value of **b** will be assigned to **big**. Finally the value of variable **big** will be printed which holds the biggest of two numbers.

Another example to find biggest of three numbers.

3. Nested if statement - To check more conditions

Using the previous type of **if** statements we can check the conditions at only one place. Alternatively we can check more conditions using the logical operators. Suppose there is a situation to check number of conditions, in different part of **if** statement, we can use the **nested if** statement. That is, the **if** statement contains another **if** statement as its body of the statement.

Format –1

```
if    (expr1)
      if (expr2)
        statement-1;
      else
         statement-2;
```

Format – 2

```
if    (expr1)
      if (expr2)
        statement-1;
      else
         statement-2;
else
         statement-3;
```

The execution form of **format-1**:

*) First the **expr1** is evaluated in both the format
*) If the result is **TRUE**, then the **expr2** will be evaluated. If **expr2** also returns **TRUE,** the **statement-1** will **be** executed.
*) If the result of **expr-2** is **FALSE** the **statement-2** will be executed.

The execution form of **format-1**:

*) As in **format-1, expr1** is evaluated first
*) if the **expr1** and **expr2** are **TRUE** then **statement-1 w**ill be executed.
*) if the **expr1** is **TRUE** and **expr2** is **FALSE** the **statement-2** will be executed.
*) if the **expr1** is **FALSE** the **statement-3** will be evaluated.

Note: *Every **else** is closest to it's **if** statement.*

Thus of statement can be used inside the body of another if statement (nesting of ifs). The following program is an example for **nested if,** to find the biggest among three numbers.

```
/* Example for nested if statement */
main ( )
{       int a,b,c;
        int big;
        printf("\nEnter three numbers : ");
        scanf("%d%d%d",&a,&b,&c);
        if (a>b)
                if (a>c)
                        big=a;
                else
                        big=c;
        else
                if (b>c)
                        big=b;
```

```
            else
                    big=c;

        printf("\nBigget no. is : %d ",big);
        getch( );
    }
```

4 Switch – Case Statement:

Whenever the situation occurs like to check more possible conditions for single variable, there will be a no. of statements are necessary. the **switch-case** statement is used to check multiple conditions at a time, which reduces the no. of repetition statements. The general format of switch-case is as follows.

```
switch (expr)
{
    case  c-1:        statement-1;

    case  c-2:        statement-2;

   case  c-4:         statement-4;

   [ default  :  statement-n;  ]
}
```

Way of Execution:

*) First the **expr** will be evaluated and it must return a constant value. The constant can be numeric or character.

*) The result of **expr** is checked against the constant values like **c-1, c-2,** etc., and if any value is matching, the execution starts from that corresponding statement. The execution will continue until the end of **switch** statement.

To avoid this continuous execution problem we can use the **break** statement. The **break** is used to terminate the process of block like switch, looping statements.

Here the **default** is an optional statement in switch. If no matching has occurred, the **default** part of statement will be executed and may occur any where in the switch statement.

```
/* Example for switch statement without break */
main( )
{
   int  a;
   printf("\nEnter  any value  for  a : ");
   scanf("%d",&a);
   switch(a)
   {
        case  1 :       printf("\nGood");
        case  2 :       printf("\nWell");
        case  3 :       printf("\nExcellent");
```

```
            default : printf("\nBad Guy");
        }
    }
```

Output:

```
Enter any value for a : 2
Well
Excellent
Bad Guy
Enter any value for a : 5
Bad Guy
```

```
/* Example for the importance of break statement */
main( )
{
   int a;
   printf("\nEnter any value for a : ");
   scanf("%d",&a);

   switch(a)
   {
        case 1 : printf("\nGood"); break;
        case 2 : printf("\nWell");  break
        case 3 : printf("\nExcellent"); break
        default : printf("\nBad Guy");
   }
}
```

Output:

```
Enter any value for a : 1
Good
Enter any value for a : 5
Bad Guy
```

Upon using the break statement the statement corresponding to the matching case is only executed. The last statement or default statement is not in need of **break**, because there is no more statements for further execution in the switch.

Looping Statements:

The simple statements that we have discussed so far are used to execute the statements only once. Suppose a programmer needs to execute the specified statements multiple times. Here comes the looping statement, which overcomes the no. of times. For example, to print the string **"Good"** five times, we have to use five individual **printf( )** statements. But, imagine if the no. of time increases to print 1000 times or N times. By using the looping statements a statement or set of statements can be executed repeatedly.

```
main( )
```

```
{
        for(i=1;i<=100;i++)
            printf("\nGood morning");
}
```

Just think about the no . of statements in the above programs, with and without looping statement. So, we can use the looping statement to reduce the size of the program.

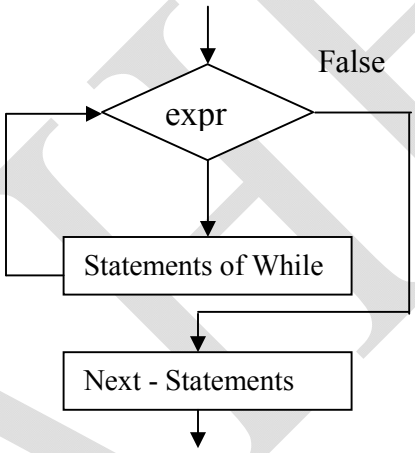
The different types of looping statements are

1. While statement
2. Do-While statement
3. for statement

1 while statement:

While is an **entry controlled** looping statement used to execute it's body of the statement any no. of times. The general format of while statement and its flowchart is as follows .

```
while ( expr)
{
  //Statmts of while;
}
  next-statement;
```



Way of execution:

- First the **expr** is evaluated, which will yield **TRUE** or **FALSE** result.
- If the condition is **TRUE,** the control enters inside the **statements of while** and after completion of these statements once again the condition will checked with new value for the next execution.
- Entry of the loop will be determined by the condition, so it is also called entry controlled looping statement.
- So the **Body of while** is executed until the condition becomes **FALSE.**
- If the condition is **FALSE**, execution jumps to the **next-statement** after the **while** statement and continues the execution.

The following example prints the numbers from 1 to n using **while** loop.

```
/* Example for while loop */
main( )
{
        int i=1,n;
        printf("\nHow many numbers");

        scanf("%d",&n);
        printf("\nThe numbers are ");
        while(i<=n)
        {
```

```
            printf("%5d",i);
            i++;
          }
}
          /* Reverse the given Integer number */
main( )
{
          int n;
          printf("\nEnter a number    :");
          scanf("%d",&n);
          printf("\nReverse of number :");
          while(n != 0)
           {
                    printf("%d",n%10);
                    n=n/10;
           }
}
```

2 Do-While statement

It is also a looping statement to execute the specified statements repeatedly (or) any number of times. In the **while** loop, condition is checked first, and execute it's statement only when the result is **TRUE**.

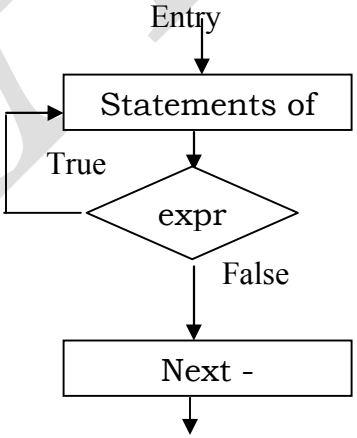
What is the difference between while & do-while?

while is a entry controlled looping statement and the statement of **while** will be executed only when the condition is **TRUE**. But in **do-while**, statement part will be executed first then only the condition will be checked. So the condition may be **TRUE / FALSE,** but the statement part will execute at least once.

The general format and it's flowchart is given below.

```
do
{
   Body of   loop;
}
while(expr);

next-statement;
```



Here, the Body of loop will be executed first and the expression **expr** will be checked after the execution of the statement parts. If the result of **expr** is **TRUE** the control starts its execution once again from Body of loop. This process will continuous until the result of **expr** is **FALSE**.

The following program is like a menu selection program.

```
/* Example for do-while looping statement */
main( )
{int ch;
        do
        {
                printf("\n1. Add\n2. Sub");
                printf("\nSelect a choice");
                scanf("%d",&ch);
         }
         while(ch<3);
 }
```

Output :

```
1. Add
2. Sub
Select a choice  1
1.  Add
2.  Sub
Select a choice  3
```

The following is another example for Decimal to Binary conversion.

```
/* Converting Decimal no. To binary  */
main( )
{
        int a,n,s=0,i=1;
        printf("\nDecimal No . :");
        scanf("%d",&n);
        printf("\nBinary  No . :");
        while(n)
        {
           printf("%d",n%2);
           n=n/2;
        }
}
```

Output:

```
Decimal No . :5
Binary  No . :101
```

3 For statement – Flexible looping statement

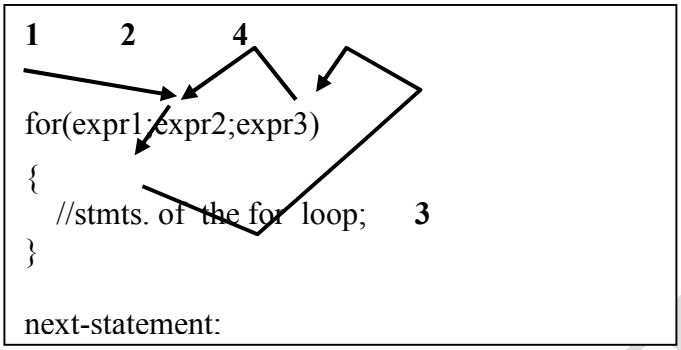
It is also a looping statement to execute the specified statements repeatedly in a simplified format than the previous loops.

In general all the looping statements have the following three steps (Parts) in a loop

1. Initialize the loop control variable
2.Check the condition whether TRUE / FALSE
3. Modify the value of the loop control variable for next execution

In the previous type of looping statements, these statements are kept separately. But in **for** statement all the three parts are kept in one place.

The general format and flowchart is given below.



Where

*) **expr1** is used to initialize the value for loop control variable
*) **expr2** is used to check condition
*) **expr3** is used to modify the value

The **expr1,** step 1 is the statement executed first and only once in the looping statement. The steps 2,3 and 4 will be executed continuously until the condition becomes false, at 2nd place. 2,3,4 is the sequence of execution in the for loop.

Way of execution:

*) First the value of loop control variable is initialized by **expr1.**
*) Next the condition is checked by **expr2** and if it is **TRUE** then the **body of loop** will be executed otherwise the control passes to the **next-statement** of the program.
*) For every **Body of loop** execution, **expr3** will be executed to modify the variable's value.
*) The above process continue until the **expr2** will becomes **FALSE**.

Examples:

1. for(i=1;i<=100;i++) /* Increasing */
{ body of loop }

i value is initialized to **1** and for every execution **i** value is incremented by **1** . So body of loop will execute **100** times.

2. for(i=100;i>0;i=i-2) /* Decreasing */
{ body of loop }

First value of **i** is initialized to **100** and for every execution **i** value is decremented by **2.** So body will execute **50** times.

The following program is used to calculate Factorial value for a given number **N.**

General formula : **n!** = 1 * 2 * 3 * 4* …….*n

To do so, any of the looping statements can be used. The initial value of the loop is 1, next increment is 1 and the summation.

```
/* To find the factorial of a number */
main( )
{
 int n, i, f=1;
```

```
        printf("\nEnter a number");
        scanf("%d",&n);
        for(i=1;i<=n;i++)
                f = f * i;
        printf("\nFactorial of %d = %d ",n,f);
}
```

One more example, following is a program generates a *fibonacci series*. The series like as

0 1 1 2 3 5 8 13 . . .

Generally the number is obtained by summing up of previous two numbers. So, first initialize **0 , 1** to **a , b** respectively and find the number as c = a + b.

Next reassign the values of **a** and **b**, like b→ a, c→b and proceed the same way for N number of times.

```
        /* Fibonacci Sequence generation */
main( )
{
        int a=0,b=1,c,n,i;

        printf("\nHow many numbers :");
        scanf("%d",&n);

        printf("\nFibonacci Sequence \n");

        for(i=1;i<=n;i++)
          {
          c=a+b;
          printf("%d\t",c);
          a=b;  b=c;
         }
 }
```

Additional information about the for loop:

The for loop has three parts inside a set of parenthesis and each is separated by the semicolon (**;**).

The **expr1** may be placed before the for loop as

```
i=10;
for(;i<100;i++)
        {  }
```

We can have more than one statement in the place of **expr1,** which are separated by commas (**,**). For example

```
for( a=4,i=0;i<10;i++)
  {     }
```

The **expr3** may also be placed in the body of the loop as .

```
for(i=10;i<100;)
        {  i++;  }
```

If any expression in the **for** is missing, the semicolon must be placed.

```
for(i=0;i<100;)
        { i++; }    /* expr3 is empty in for loop /*
```

The following loop is used execute the statements indefinitely, because there is initialization, condition and modification.

```
for( ; ; )
   {    }
```

More than one condition statement can be used in the **expr2** place of **for** loop as

```
for(i=1;i<10&&j<20;i++)
        {   }
```

Nested for loop:

Just like a nested **if,** nested for loop is also possible. In the nested for loop, the statement part of the loop contains another for statement.

```
for(expr1;expr2;expr3)  ——— O
                        ——
  for(expr4;expr5;expr6)     I

    {

      // body of loop
```

Where **O** is an outer loop
I is an inner loop

Way of execution:

For every value of outer loop, the inner loop will execute no. of times. In the nested loop, the body of loop will be executed until both the expressions **expr2** and **expr5** becomes **FALSE**.

For example

```
for(i=1;i<=10;i++)
for(j=1;j<=5;j++)
        printf("\nIndia");
```

Here for every value of **i** the **j** loop execute the statement part **5** times. So totally the string **India** will be printed **50** times (10 * 5 = 50).

The following example shows a clear output for you about the nested loop.

```
/* Example for nested for loop */
main( )
{
        int i,j;
        for(i=1;i<=10;i++)
          for(j=1;j<=5;j++)
                printf("\ni = %d     j  = %d   ",i,j);
}
```

Output:

```
i = 1     j  = 1
i = 1     j  = 2
.
i = 2     j  = 1
```

```
.
.
i = 10    j = 5
```

Here for every value of **i** the **j** loop executes **5** times.

Break Statement - To stop the process

The looping statement is used to execute a statement repeatedly for specific no. of times, but the user cannot exit from the looping in middle. The **break** statement providing a facility to exit from the loop whenever we in need.

The **break** is mostly used in loops like **for, while, do-while** and **switch** statements. When it is used in the program it terminates the execution of the block or jumps from the current block to the next.The following program sums the positive numbers only and when user gives negative number the program break the process.

```
/* Example for break statement */
main( )
{
 int n,s=0;
 printf("\nEnter numbers one by one –ve to stop\n");
 do
 {
  scanf("%d",&n);

  if (n>0)
        s=s +n;
  else
        break;
  }while(1);
 printf("\nSum    =  %d  ",s);
}
```

Output:

```
Enter numbers  one by one  and –ve to stop
3
5
1
-4
Sum    =   9
```

The break statement is used to terminate the execution process from the block where it is specified.

```
/* Example for break statement  */
#include <math.h>
main( )
{
        int i,j;
        for(i=1;i<=5;i++)
           for(j=1;j<=5;j++)
```

```
                    if (j%2= =0)
                            break;
                    else
                    printf("\ni= %d  - j=%d",i,j);
        }
```

Output :

```
        i= 1  - j=1
        i= 2  - j=1
        i= 3  - j=1
        i= 4  - j=1
        i= 5  - j=1
```

Here, when value of **j** is **2**, the expression **j %2** is **0**. So the control will exit from the inner loop (ie **j** loop) and not from both. So, there is no chance to execute the **j** loop with more than value **2.**

Continue Statement

The continue statement is used to continue the process of execution by skipping some of the statements placed after the statement itself. It is usually used in the looping statements.

Generally a loop will execute all the statements in it. But, instead of doing so, we may skip some of the statements according to the condition and we can continue from the beginning of the loop with next iteration value.

For example, a mark list preparation program may be used to read details of N students. We can use the looping statement to read the details about n students. If few students of the class had left (for example, rollno's 10,15 and 30), how can we get details except numbers 10,15 and 30? The solution is using the **continue** statement as follows:

```
   for(i = 1; i < 60; i++)   // Assume 60 Students
     if (( rollno = =10)||( rollno = =15)||(rollno= =30))
            continue;   // Continue with next student
        else
         {
           // Read the details of students other than left
         }
```

Another example is printing all the numbers from **1** to **n** except the number indivisible by **3**. For this program when **(i%3 == 0)** the number must not be printed and the execution must continue with the next I value.

```
          /* Example for continue statement */
          main( )
          {
                int n,i;
                printf("\nHow many no.");
                scanf("%d",&n);
                for(i=1;i<=n;i++)
                    if (i%3= =0)
                            continue;
                   else
                            printf("%d\t",i);
```

```
    }
Output:
        How many no. 10
        1    2    4    5    7    8    10
```

Here for every value of **i** the condition is checked. If the value of **i** is divisible by **3**, the print statement is skipped and the execution continues with next number.

Possible Questions

Part-B (2 marks)

1. What is an identifier?
2. Say which of the following are valid C identifiers:
   1. Ralph23
   2. 80shillings
   3. mission_control
   4. A%
   5. A$
   6. _off
3. Write a statement to declare two integers called i and j.
4. What is the difference between the types float and double.
5. What is the difference between the types int and unsigned int?
6. Write a statement which assigns the value 67 to the integer variable "I".
7. What type does a C function return by default?
8. If we want to declare a function to return long float, it must be done in, at least, two places. Where are these?
9. What is the difference between getchar( ), fgetchar( ), getch( ) and getche( )?
10. List the role of a field-width specifier in a printf( ) function.
11. If a character string is to be received through the keyboard which function would work faster? Why?

Part-C (6 marks)

1. Give the steps to compile and execute a C program.
2. With syntax and example explain all the different forms of if statement.
3. Explain in detail about various types of operators. Provide examples for each.
4. What is while loop? Give syntax. Explain it with an example program.
5. With syntax and example explain the character Input/Output methods with example.
6. What are the functions used in Formatted I/O? Explain in detail with examples.
7. What is the use of switch statement? Give its syntax and explain with an example.
8. Write a program to calculate factorial of a given number.
9. Write a program print the fibonacci series.

KARPAGAM ACADEMY OF HIGHER EDUCATION					
PART - A (ONLINE EXAMINATION)					
MULTIPLE CHOICE QUESTIONS (Each question carries one mark)					
SUBJECT: PROGRAMMING FUNDAMENTALS USING C/C++					
QUESTIONS	OPTION1	OPTION2	OPTION3	OPTION4	**ANSWER**
. _____ refers to finding value that do not change during execution of program.	keyword	identifier	constant	token	constant
_____ constant contains single character enclosed within pair of single quote marks.	string	variables	character	numeric	character
_____ is data name that may be used to store a data value.	string constant	variables	character	numeric	variables
Characters are usually stored in _____ bits.	8	16	24	32	8
Floating point numbers are stored in _____ bits.	8	16	24	32	32
_____ imparts fixed meaning to the compiler and these meanings cannot be changed.	variables	constant	keywords	functions	keywords
_____ operator is used for manipulating data at bit level.	logical	bitwise	arithemetic	sizeof	bitwise
_____ header file should be included for calling math function in source program	conio.h	stdio.h	math.h	stdlib.h	math.h
Reading a single character can be done by using _____ function.	getchar	putchar	gets	putch	getchar
An arithmetic expression will be evaluated from _____ using rules of precedence of operation.	left to right	right to left	top to bottom	bottom to top	left to right
Execution of C Program begins from _____ function.	user defined	main	header file	statements.	main
Every C program must have exactly _____ main function	one	two	three	none	one
_____ function is predefined and is for printing output	scanf	printf	getch	putch	printf

The lines beginning with /* and ending with */ are known as lines.	document	comment	definition	declaration	comment
_____ header file should be included for calling mathematical function in source program	conio.h	stdio.h	math.h	stdlib.h	math.h
_____ function writes character one at a time to terminal	getchar	putchar	getch	puts	putchar
C Language supports _____ logical operators	2	3	4	5	3
_____ lines are not executable statements and are ignored by the complier	function	comment	main	none	comment
Every statement in C should end with _____ mark	:	;	dot	none	;
_____ is new line character	\b	\n	\d	\l	\n
_____ cannot be used as variable name	constant	keywords	operators	function name	keywords
Binary operators takes _____ no.of operands	1	2	3	4	2
_____ imparts fixed meaning to the compiler and these meanings cannot be changed	variables	constant	keywords	functions	keywords
In variable declaration, first _____ characters are treated as significant by compilers	8	10	31	32	8
The C programs are written only in _____	lower case	upper case	title case	sentence case	lower case
The variables are initilaised using _____ operator	>	=	?=	+	=
Which is the incorrect variable name?	else	name	age	char4	else
Which is the unary logical operator?	&&	||	!	all the above	!
The input function scanf can use _____ format specification to read in string of characters	%c	%s	%d	%l	%s

printf function is used with _____ format to print strings to the screen	%c	%l	%s	%p	%s
In ASCII character set the uppercase alphabet represent codes _____	65 to 90	96 to 45	97 to 123	1 to 26	65 to 90
_____ statement tests value of a given variable against list of case values	switch	if..else	for	while	switch
_____ statement causes exit from switch statement	switch	break	goto	end	break
_____ operator takes 3 operand for making logical decisions	+	?:	=	:?	?:
The _____ is powerful branching statement used to control the flow of execution of statements	if	for	goto	while	if
_____ is two way decision making statement and is used in conjunction with an expression	if	switch	goto	while	if
In if condition the statement block below is skipped if the value of the expression is	TRUE	FALSE	one	zero	FALSE
In C multiway decision statement is _____	while	if..else	switch	for	switch
Variables declared inside for loop is called_____ variables	Constants	loop control	data control	data	loop control
Conditional operator is a combination of ___ and _____ operator	?:	&?	&*	?,	?:
In conditional operator testexp?exp1:exp2, if the testexp is non zero is evaluated	exp1	exp2	exp3	exp4	exp1
In conditional operator testexp?exp1:exp2, if the testexp is zero is evaluated	exp1	exp2	exp3	exp4	exp2
Case Labels end with ____ operator	:	;	.	,	:
In switch statement if the value of the expression does not matches with any of the case values is	optional	case	default	loop	default
While loop is _____ statement	entry controlled	exit controlled	branching	none of the above	entry controlled

Do.. While loop is _____ statement	entry controlled	exit controlled	branching	none of the above	exit controlled
In _____ loop tests is performed at the end of body of the loop	entry controlled	exit controlled	branching	none of the above	exit controlled
In _____ statement body of the loop is executed first	do..while	else	for	none of the above	do..while
In for loop body of the loop is executed if the test condition is	TRUE	FALSE	zero	one	TRUE
In for loop _____ operator is used separate the sections	;	:	.	none	;
In for loop three sections enclosed within parenthesis must be separated by	colon	semicolon	comma	dot	semicolon

UNIT II

Syllabus

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.

FUNCTIONS

Introduction

A Program is a collection of instructions and in some cases, the program may have repeated statements. If the number of repeated statement is one or two, then it does not make any problem in the program. Suppose the number of repeated statements is more in a program, it will automatically increase the size of the program, unnecessarily. So, how do we reduce the size of the program? It can be done by writing these repeated instructions in a separate program. These programs can be utilized whenever needed. That separate code/program is called as a function. Function may also be named as procedure. The functions are of two types

1. Library functions.
2. User defined functions.

Library function- A Readymade one

This is a special type of function, which is pre-written and present along with the compiler itself. For example, abs( ), sqrt ( ) etc are library functions and we have discussed about them in the second chapter. The drawback of library function is that, it has restricted operations and it is not allowed to alter the existing functionality.The library functions are just like our readymade dresses. The drawback of ready made is that it will not match every ones need, but can be used on suitable occasions.

User defined functions – Design your own:
The user can write a function according to their wish and requirements and this type of function is called as user defined function. It is just like designing a dress depending upon one's taste. So if you are not satisfied with the readymade, design your own.
The purpose of having a function in a program is to reduce the size of the program and in some cases this can also be achieved by using the looping statements. The following is a program and how the same is reduced is illustrated.

```
main()                                  }}
{                                       }}
        printf("\nHello");
        printf("\nGood morning");    Set 1
        printf("\nHello");          Set 2
        printf("\nGood morning");
        printf("\nHello");
        printf("\nGood morning");
        printf("\nHello");
```

```
        printf("\nGood morning");
        printf("\nHello");
        printf("\nGood morning");
    printf("\nHello");
        printf("\nGood morning");
    }
```

Can we reduce the size of the above program? Yes. We can. The following is a revised version of the program using for looping statement.

```
/*  Minimized program using for loop */
main()
{       int i;
        for(i=1;i<=3;i++)
        {       printf("\nHello");
                printf("\nGood morning");
        }       }
```

Suppose the repetition occurs in different part of the program instead of continuous one, using looping is not a solution. Here comes the function. Yes. Keep the repeated set of statements in a separate part of the program (some time called as sub-program). Whenever these repeated statements are required, the sub-p voked and

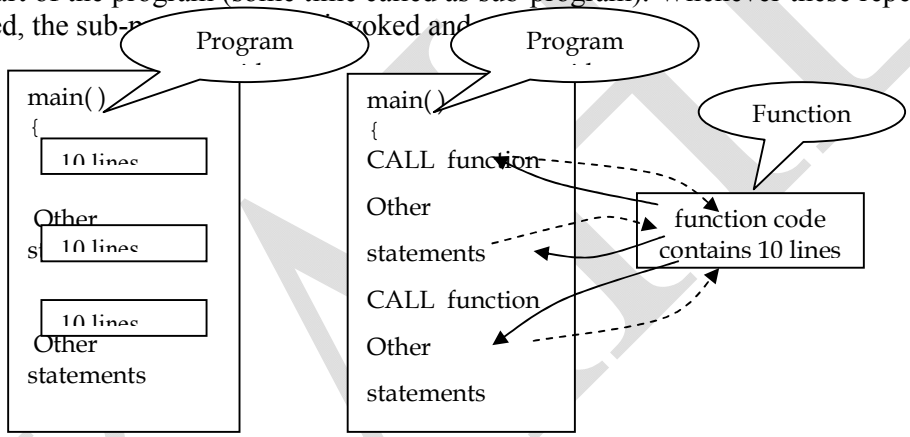
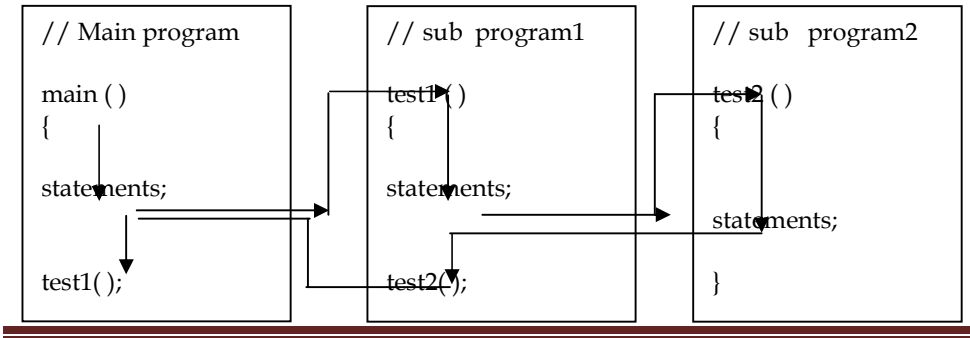


Figure a Figure b

Figure (a) is a program with repeated code in three places and the aim of all the 10 lines are same. In figure (b) the repeated statements are available as a sub-program / function. This function can be invoked/called wherever the repeated code is necessary.

In the figure (a) the repeated code is 10 lines, totally it occupies 30 lines, because it is in 3 places (3 * 10 = 30). But in the figure (b) instead of 10 lines, only one instruction (CALL) is used to invoke the sub program. Totally 30 lines of the figure (a) is reduced into 3 lines and 10 lines in the function. This function can be invoked any number of times at any place.

The following diagram is illustrating how the functions are being invoked and processed with multiple functions.



In the above diagram there is two sub programs namely test1 and test2.

- First we know the main( ) function starts its execution and it calls the user defined function test1( )
- Before starting process, the status of main( ) is stored into stack and execution continuous in test1( )
- The function test1( ), in turn calls another function test2( )
- As in the previous case status of test1( ) is pushed in to the stack.
- Now stack contains status both test1( ) function and main ( )
- The execution continuous in test2( )
- After completion of test2( ), the control is returned to the test1( ) and continues the execution until the end of the function
- When test1( ) is completed, the control returns to main( ) program and once again the process is resumed.

General format for function declaration:

```
Return-type function-name(arg1,arg2...)
     {

       Local variable declaration

       Body of the function

       [  return; ]
   }
```

Here

- ⇨ Return-type is type of data returned by the function
- ⇨ Function-name is the name of the function
- ⇨ arg1 , arg2 … are parameters of the function

Look the example for a function declaration

```
int swap(int a,  int b)
{
        // Body of the function;
    }
```

The first int is the type of data to be returned by the function swap( ) and a, b are the arguments to the function.

What is parameter / arguments?

A variable, that is used to carry the data to the function, is called as parameter or argument. The parameter may be either a value parameter, carrying value directly or a variable parameter carrying values using the variable.

In the example the arguments a, b are belong to the same data type. The function declaration with the parameter is as follows.

```
int swap(int a,  int b);
    /* a & b  must be declared individually */
```

The arguments of the function can also be declared as shown below.

```
int  swap(a , b)
int a, int b;
{
  /* Body of the function;  */
       return ( 5 ); /* Returns Integer Value */
}
```

This type of declaration is called K-R declaration. (K–Kernighan and R–Ritchie, who are the authors of C programming language)

In case of normal variable declaration, number of variables of same type may be declared by single declarative statement. Even though the variables are of same type, we can't declare as above said in a function.

```
int    swap(int a , b);  /*  Error  */
```

Note: The arguments must be declared individually even if they belongs to the same data type

How to call the function?

We used to call persons by their names. Here also the function is invoked by specifying the function-name with required parameters. The following is an example program for function call with parameters.

```
/* Simple example for function definition and calling the same
*/
        main( )
        {
     display( 10 );
        }
                        Function calls with 10
        display  (int  a)
        {       printf("a = %d ",a);   }
```

The main( ) function calls the user defined function namely display( ) with the value 10 and it is an argument to the function. Now the value 10 is carried and assigned to the variable a. It is actually an assignment statement like a = 10.

Return statement:

This is a simple statement used to return the control to the calling function from the called function and this is required in some cases. The return statement can be used to return a value from called function to the calling function. The format of return statement is

```
return (expression);
```

- ⇨ It is an optional statement in the function
- ⇨ It may occur more than once in the function
- ⇨ It may appear anywhere in the function.

Let us see some of the possible usage of return statement

1. return(5) => Return value 5 to the calling function
2. return (a+b) => Result of expression a+b will be returned
3. return ('a') => Returns the character 'a' to the calling function
4. return (&a) => Returns address of 'a'
5. return => Returns the control to the called function.
   (It is used to stop and return explicitly)

Look the following program, which is a complete program illustrating the function and how it has been invoked.

```
/* Complete example for a function */
main( )
{  int  a =10, b=20,c;
     c = sum(a,b);
  printf("\nSum = %d ",c);
}
int  sum(int  x, int y)
{   return (x+y);}
```

Here we are passing the value of a and b to the user defined function sum() and this function compute the sum of these two number s and return the result to the calling function.

Types of function:

The functions are of four types, classified based on the parameter passed to the function and value returned by the function. They are functions with

1. No argument & No Return
2. No argument & Return
3. Argument & No Return
4. Argument & Return value

No argument and No return:

In this type of function, nothing is passed to the called function from the calling function and also nothing is returned from the called function to the calling function. The following is an example for this

```
// Main program
main( )
{
  line( );
  printf("Welcome to all");
  line( );
}
```

```
//User defined function
void   line( )
{
  int i;
  for(i=1;i<50;i++)
      printf("-");
}
```

In the above example, we are not passing any value to the function line( ) and it does not return any value to the called function. The aim of the function is very simple that it prints a line. The void is a data type, which represents returning NULL data and here the function returns nothing.

Default return type of the function is **int**eger

No argument and return value:

In this type of function the user may define a function to return some value without passing anything. This type of function does not take any argument to the called function, but it returns value to the calling function.

```
main( )
{
  int  a, b, c;
  a = input( );
  b = input( );
  c = a+b;
  printf("Sum = %d ",c);
}
```

```
int   input( )
{
   int  x;
   printf("Enter a value ");
   scanf("%d ",&x);
   return (x);
}
```

Here the function input ( ) is used to read an integer value without the scanf( ) statement in the calling program. We can use it when we are in need of any integer value. This function does not take any argument but it returns an integer value to the calling function.

Argument and No return value:

This type of function receives an argument but it does not return any value to the calling function.

```
// Main program
main( )
{
  line( );
  printf("Welcome to all");
  line(15 );
}
```

```
//User defined function
void   line( int  n)
{
   int i;
   for(i=1;i<n;i++)
      printf("-");
}
```

The line( ) function receives an argument as no. of characters (-) to print, here 15 times and they are printed as line. But this function does not return anything to the main ( ) function.

With argument and a Return value:

This type of function receives an argument and also returns value to the calling function.

```
main( )
{
  int  a =10, b=20,c;
  c = sum(a,b);
  printf("Sum = %d",c);
}
```

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

The function sum( ) takes two integer values as arguments and returns the sum to the calling function.

The passed values are only the photocopy of a and b. If any modification is made to this value inside the function sum( ), it does not affect the main ( ) function's a , b values, as the modifications in photocopy does not affect the original documents.

Normally a function can't return more than one value

In functions the changes in the local variable does not affect the arguments of the calling function value. The following example will illustrate this.

```
/* Example for passing values to the function and result of changes */
main( )
{  int  a=10;
  printf("\nBefore passing  : %d ",a);
   disp(a);
  printf("\nAfter  passing  : %d ",a); }
void disp( int  x)
{  printf("\nValue inside the function : %d ",x);
  x=100 ;
  printf("\nValue inside the function : %d ",x);}
```

```
Before passing  : 10
Value inside the function  : 10
Value inside the function  : 100
     After passing : 10
```

In main( ) program variable a contains the value 10. The function disp(a) is called with the value of a. Now the copy of a is passed (assigned as x=a) to the function disp( ). Inside the function

disp( ), x contains the value of a. In the function value of x is changed as 100, this change affects only the local variable x and note that the value of a in main( ) program remains the same.
Note:
The pointer will help to solve these kind of problems such as modification should reflect everywhere and returning more than one value etc.

Same Variable Name - No Problem
Name of the variable is an identifier and it is only for the user's reference. System uses its own way to access the values and to identify it. So the variable name in one function may occur in any other function or in the block also with the same name.

```
/* Illustration of same variable in the program */
main( )
{
  int  a=10;
  disp(a);      /* a is local  to  main( )  */
}
void disp( int  a)
{
 a=100;       /*  a  is  local  to disp( )  */
 printf("\na  =  %d ",a);
}
```

In the above program, the variable a occurs in both main( ) function and user defined function disp( ). Though the variable name seems to be identical, their memory allocations are different. The variables are individual components of each of the function and it will not confuse the compiler.
The following program illustrates the above discussion, about the different addresses and it solves the problem of same variable.

```
/* To prove the memory allocation is vary even the  names of the variable are same. */
main()
{
  int a;
  printf("\n%Address of a in main     :  ",&a);
  test();
}
void test()
{
 int a;
 printf("\nAddress of a in function  : %u",&a);
}
```

```
    Address of a in main        : 23344
        Address of a in function    : 24444
```

The output of this program is the memory address of two variables, which are equal. But the memory address is not same. So this is not a problem in a program.

The following is a program to find maximum of two numbers using function

```
/* To find the maximum of two using function */
main( )
{   int a ,b, big;
```

```
    printf("\nEnter two numbers : ");
    scanf("%d%d",&a,&b);
    big = max(a,b);
    printf("\nBiggest no is = %d ",big);
}
int   max(int x, int y)
  {  int temp;
  temp = x>y ? x : y;
    return (temp);
  }
```

```
Enter two numbers : 10 30
Biggest no is = 30
```

The calculation of nCr value is a very good example using function. We have already discussed it theoretically. In this program the function fact( ) is used to calculate the factorial value of a given number.

```
    /* To find nCr program using function */
main( )
{
  int n, r;
      float ncr;
  printf("\nEnter the value of N  &  R : ");
      scanf("%d%d",&n,&r);
  ncr =(float)fact(n)/(fact(n-r) * fact(r));
      printf("\nnCr  value  =  %5.2f  ",ncr);
    }
int fact(int m)
    {
     int i,f=1;
 for(i=1;i<=m;i++)
          f =f * i;
    return f;
   }
```

```
        Enter the value of N  &  R : 5    2
nCr  value  =  10.00
```

If function is not used in the above program we should write separate set of statements to calculate the 3 different factorial values, as we have seen about this in the starting of this chapter.

While using function, to calculate n! value, just the statement fact(n) is enough. So fact(n) gives n! , fact(n-r) give (n-r)! and fact(r) gives the r!. It reduces the no. of instructions and memory automatically.

The following is an another example to implement the pow( ) function.

```
  /* To compute x^n without using library function */
  /* or Implementation of power() function  */
main( )
{
int n;
float x, s;
```

```
printf("\nEnter the values of X & N ");
scanf("%f%d",&x,&n);
s = power(x, n);
printf("\nPower(%5.2f , %d ) = %5.2f",x,n,s);
}
float power(float x, int n)
{    int i,s=1;
   for(i=1;i<=n;i++)
          s = s * x;
   return (s);}
```

```
Enter the values of X & N 10 3
Power(10.00 , 3 ) = 1000.00
```

Problems in return type – Take care:

The return type of the function should be specified carefully. If it is not proper the value may be converted into some other format and result may go wrong. We should be careful while returning a real value. The following program tells the importance of the return type specification.

```
/* Program without specifying return type */
main()
{
  float a;
  a=test();
  printf("\nResult of function calling : %f",a);
}
test()
{
return(2.5);
}
```

No return type is specified, by default it is an integer

Output:

```
Result of function calling: 2.000000
```

The return type of the function is not properly defined here. In C, the default return type is integer. Here the function returns the value as 2.5, using the return statement, which is actually a float value. But the return type is not matching with it and the float value is converted into integer value. Finally we are getting the unexpected result as 2.000000. If we add the correct return type in the function declaration, the modified program as follows.

```
/* Program with correct return type */
main()
{
  float a;
  a=test();
  printf("\nResult of function calling : %f",a);
}
float test()
{
return( 2.5);
}
```

Return type is specified properly as float

Output:

Result of function calling: 2.500000

Now we got the correct result from the function. So the return type of the data from the function should be considered important.

Calling with expression:

The parameter to the function may be a value or variable or it may be an expression. The function can be called by value is by directly giving the value as a parameter to the function. Following is a program to illustrates the function call by giving the values directly.

```
/* Calling a function with by value   */
main( )
{
  display( 10 );
}
display ( int a)
{
 printf("a = %d ",a);
}
```

Function calling with value 10
10 is assigned to the variable a

Here the function disp( ) is called directly with value as 10. The other way to call the function is by using a variable. In this case instead of value, a variable is used as a carrier of the value.

```
/ * Calling a function with variable   */
      main( )
      {
        int a =10;
        display( a );
      }
      display  (int  b)
      {
        printf("b = %d ",b);  }
```

Function calling with a

One more way to call the function is by passing an expression. In this case result of the expression will be passed as argument to the function. The example below illustrates the function call done using an expression.

```
 /* Calling  by   Expression   */
main( )
{  int  a=5 , b=10;
  display( a + b );
}
display (int c)
{
   printf("c = %d ",c);
}
```

In this program the result of expression 'a+b' is passed as argument to the function display( ). The value of a is 5 and b is 10 and therefore a+b is 15. The result 15 is passed to the function display( ) and received by the function argument c. (ie c = a + b).

Passing Array to the function:

What we have discussed so for is passing and returning one or more simple values. This is not enough for us in all the applications and we may require passing the arrays to the function. Now let

us have a view on, how to pass the array value to the function. The following is the simple example for passing array values to the function.

```
/* Example for passing array to the function */
main( )
{   int i, n, a[15];
    printf("\nNo. of elements :");
  scanf("%d",&n);
  printf("\nEnter %d values\n",n);
   for(i=0;i<n;i++)
          scanf("%d",&a[i]);
 printf("\nYour values are \n");
  for(i=0;i<n;i++)
           display(a[i]);}
display(int m)
 {        printf("%d\t",m); }
```

Recursive function:

In some cases the function invokes itself, whenever the function calls itself, this special type of function is called as recursive function. The following example is the simple recursive function calling.

```
        /* Function calling itself or recursive call */
display( )
{          printf("\nWelcome");
 display();    /*function calls itself or recursive */}
```

The calling of function display( ), will execute the instructions, within it and one of the instruction is display( ) (ie statement for invoking itself). If the function is some where else, the control will be transferred as usual, but here the function display( ) calls itself. Therefore for every execution the function is called indefinitely. The result of the above program is

Welcome
Welcome

Here there is no chance of terminating the function from execution. To avoid this indefinite execution we have to use conditional statements to stop the recursion.

A very good example for recursive function is finding factorial of a number. The factorial value of a number is calculated by any one of the two methods given below.

i) n! = 1 * 2 * 3 * …….* n

(or)

ii) n! = n * (n-1) !

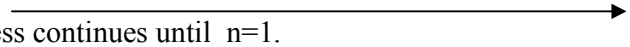
First method of calculation has been discussed already and the second method needs a different approach.

As a part of this expression we have to calculate (n-1)! which is again similar to calculating n! value. This process will continue until the value of n becomes 1. If the value of n is 1, the function returns to the calling function.

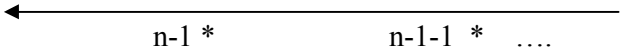
n! = n * (n-1) !

(n-1) ! = (n –1) * (n-1-1) !

(n-1-1)! = (n-1-1) * (n-1-1-1) !



process continues until n=1.



n! = n* n-1 * n-1-1 * ….

Example : 3 ! = 3 * (3-1) !

2! = 2 * (2-1) !

1 !
Now n is 1. So it returns the value 1.

3 ! = 3 * 2 * 1
= 6

Recursive function is just like a looping statement and the termination from the recursion is achieved using conditions.

STRINGS

Introduction

What is a String? A String is a collection of characters that are enclosed within double quotes (**"**). String can also be also called as a **character array**. Inside the double quotes any acceptable character from the character set can be present. To store a single character we have to use the **char** data type. But there is no data type to store a string of characters. This problem is overcome by declaring the group of characters in an array, using **char** data type. Therefore a simple string is declared in one-dimensional array. The format for declaring the string variable is as follows.

```
char variable [ size ];
```

- **char** is the data type of the variable
- [illegible]en to the array.
- [illegible]o. of characters that can be stored in the array.

Example:

```
char str[15];
```

- Here **str** is the name of the array variable name and it can store a maximum of 15 characters.

What is end of a string?

The strings should end with a NULL character **'\0',** an escape sequence character. It is used to identify the end of a string. The ASCII value of this character is **0** (Zero). NULL value is assigned automatically to the end of string. It can also be inserted explicitly by the programmer.

Assigning Values to the string variable:

The values to the string can be assigned in any one of the formats shown below.

1. char str[10] = {'K','A','R','T','H','I'};
   =>In this one dimensional array, values are assigned character by character to the string variable **str**.
2. char str[10] = "KARTHI";
   =>In contrast to the above mentioned character by character assignment, entire string can also be assigned to the string variable **str**.
3. char str[ ] = "Coimbatore";
   => Here, the size of the string variable **str** is automatically assigned to it depending on the no. of characters. Note, that the size of the array is not declared here. Memory allocation is done automatically based on the number of characters assigned to the variable.

The following diagram shows the values and its memory location

char name[ ] = "KARTHI";

Location →	0	1	2	3	4	5	6	
Character →								

From this figure we may conclude

character K is stored at name[0]
character A is stored at name[1]
character R is stored at name[2]
character T is stored at name[3]
character H is stored at name[4]
character I is stored at name[5] and
the end of string is at name[6]

How to read string?

Characters of the string is read by using the **%s** format specified for usage in the **scanf( )** function. If **scanf( )** function is used, blank space should not be present as a part of the string. At this situation we can use **gets ( )** function to read string with blank space.

*) Otherwise we can use **[…]** and **[^ …]** options in scanf( )

```
/* Simple example to read and display a sting */
#include <string.h>
main( )
{
  char str[15];
   scanf("%s",str);
   printf("\nString : %s ",str);
}
```

The statement **scanf("%s",s)** is used to read a string with the format string **%s**. Here the array name refers to the base address of the string or character array **s**.

If there is a blank space in the input string the **gets(s)** statement can be used. If the input string need to include the new line character, we can use the format string as **["%[^\n]"].**

The string can be used to process individual characters as follows.

```
/* To process character based */
main( )
{
        char s[15];
        int i=0;
        scanf("%s",s);
        printf("\nString : ");
        while(s[i] != '\0')
                printf("%c",s[i++] );
 }
```

The while loop will execute until the end of string (ie **'\0')** and it is checked by the condition **s[i] != '\0'.** If the character present in the **i**th location is equal to **'\0',** then the loop will stop its assigned function.

Library Functions in String:

The operations like, copying a string, joining of two strings, extracting a portion of the string, determining the length of a string etc. cannot be done with arithmetic operators. String based library functions are used to perform this type of operations and they are located in the header file **<string.h>.**

Four important string based library functions are

1. strlen( ) - To find the length of a string
2. strcpy( ) - To copy one string into another
3. strcat( ) - To join strings

4. strcmp( ) - To compare two strings.

Function	Use
strlen	Finds length of a string
strlwr	Converts a string to lowercase
strupr	Converts a string to uppercase
strcat	Appends one string at the end of another
strncat	Appends first n characters of a string at the end of another
strcpy	Copies a string into another
strncpy	Copies first n characters of one string into another
strcmp	Compares two strings
strncmp	Compares first n characters of two strings
strcmpi	Compares two strings without regard to case ("i" denotes that this function ignores case)
stricmp	Compares two strings without regard to case (identical to strcmpi)
strnicmp	Compares first n characters of two strings without regard to case
strdup	Duplicates a string
strchr	Finds first occurrence of a given character in a string
strrchr	Finds last occurrence of a given character in a string
strstr	Finds first occurrence of a given string in another string
strset	Sets all characters of string to a given character
strnset	Sets first n characters of a string to a given character
strrev	Reverses string

1. Length of the string - strlen( ):

The function, **strlen( )** is used to find the length of the given string. The general format is as follows.

```
int     strlen ( str );
```

Where **str** is a string variable and it returns number of characters present in the given string.

Example:

```
char   str[10] = "Karthi";
       len = strlen(s)
```

=> It returns length of the string as **6** and it is stored in the variable **len**.

```
char  s[10] = "Welcome\0";
 len = strlen(s);
```

=> This example returns length as **7,** because it will not count NULL character ('\0') as a character of the string .

len=strlen("AB**bbb**CD ");

=> where **b** is a blank space. In this case the blank space is also treated as a character. So, length of this string is **7**. (Including blank space)

The Null character (\0) is not a countable character in the string

2 Assigning the string - strcpy ( ) :

The function **strcpy( )** is used to copy the content of one string into another. We can't use the **assignment operator** (=) to assign a string to the variable. By using this function only we can perform assignment operation on string.

```
char s[15] ;
s = "Man";   /* This is not possible in C */
```

The general format is as follows.

```
strcpy( s1, s2);
```

Where **s1** and **s2** are string variable

Here **s2** is the source string and **s1** is the destination string. After the execution of this function content of **s2** is copied into **s1** and finally both **s1** and **s2** contains the same values.

Example :

1. char s1[ ] = "Karthi";
   char s2[ ] = "Good";
   strcpy(s1,s2);
   => After the execution the content of the string **s2** is copied into string **s1**. Therefor the string **"Karthi"** is replaced with the string **"Good"**. Now both **s1** and **s2** contains the string **"Good"**.
2. char s1[10];
   char s2[10] = "Karthi";
   strcpy(s1,s2);
   => Here also the content of **s2** is copied into **s1** and both contains the string as **Karthi**.

3 Joining Strings - strcat( ) :

The function **strcat( )** is used for joining two strings. The general format is as follows

```
strcat(s1 , s2 );
```

Where **s1** and **s2** are string variables.

Here the content of **s2** is appended (or) joined to the contents of **s1**. After the execution **s1** now contains its own content, followed by the contents of **s2** and **s2** retains the same.

Example:

1. char s1[10] = "Good" , s2 [10] = "Morning";
   strcat (s1,s2);
   => After the execution of this function, **s1** contains "**GoodMorning**" and **s2** contains "**Morning**".

2. char s1[10] = " " , s2 [10] = "Welcome";
   strcat (s1,s2);
   => After execution, both **s1, s2** has "**Welcome",** because first variable **s1** does not have any character in it.

4 To Compare - strcmp( ) :

This function is used to compare two strings. To compare any numeric values we use relational operators, by using these operators we can't compare strings. The **strcmp( )** performs the function of comparison. The general format is as follows.

```
int    strcmp(s1 , s2 );
```

Where **s1** and **s2** are string variables.

This function returns any one of three possible results.

*) Result is **0** when both strings are equal. (s1 = s2)

*) Result is **Positive** value, if **s1** is greater than **s2** (s1>s2)

*) Result is **Negative** if **s1** is less than **s2**. (s1<s2)

Note:

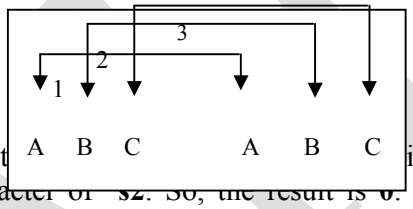
The result of comparison is obtained by calculating the difference between the ASCII value of the corresponding characters. The comparison is made by checking the corresponding characters (ASCII values) one by one between two strings.

The first character of s1 is compared with the first character of s2. If they are equal, the process passes on to the next character on the string until they meet with mismatch or no more character to process.

The process of comparison is terminated if any mismatching occurs or end of string is reached.

Example:

1. char s1[5] = "ABC";
   char s2[5] = "ABC";



strcmp(s1,s2);

=> This function returns **0** as a result, as both **s1** and **s2** contains the same characters. The characters of **s1** is equal to the character of **s2**. So, the result is **0**. (ASCII difference of these characters).

2. char s1[5] = "ABC";
   char s2[5] = "abc";
   strcmp(s1,s2);

=>This function returns **-32** as a result, as ASCII difference between **A** and **a** is **-32** (65 – 97). ASCII value of **'A'** is **65** and **'a'** is **97.**

3. char s1[5] = "ABz";
   char s2[5] = "ABC";
   strcmp(s1,s2);

=> This function returns **55** as a result. Because ASCII difference between **z** and **C** is **55**(122 – 67). ASCII value of **'z'** is **122** and **'c'** is **67.**

Here is a program to illustrate, how to read a string and find out its length without using the library function.

```
/* Implementing strlen( ) function */
#include <string.h>
main( )
{
 int i= 0;
 char str[25];
 printf("\nEnter a string    : ");
 gets(str);
 printf("\nYour given string : ");
 while(str[i])
```

```
            printf("%c",str[i++]);
          printf("\nLength of string  : %d ",i);
        }
```

Output:

```
        Enter a string    : karthi
        Your given string : karthi
        Length of string  : 6
```

```
  /*Check whether the given string is palindrome or not*/
      #include <string.h>
      main( )
      {
            int i,l,poly=1;
            char s[50];
            printf("\nEnter a string    : ");
            gets(s);
            printf("\nGiven string  is  : %s",s);
            l=strlen(s);
            printf("\nLength of string  :  %d\n",l);
            l = l-1;
            for(i=0;i<=l;i++)
                if (s[i]!=s[l-i])
                      poly=0;
            if (poly = =1)
                      printf("\n'%s'  is polindrome",s);
            else
                      printf("\n'%s'  is not polindrome",s);
       }
```

strrev( ) => This function is used to reverse the string

```
      char s[ ] =”Hello”;
      strrev(s);
```

=>Now the content of the string **s** is reversed.

A program to find out whether the given string is palindrome or not using the library function.

```
      /* Palindrome checking using library function */
      main()
      {
            char s1[15],s2[15];
      clrscr();
            printf("\nEnter a string : ");
            scanf("%s",s1);
            strcpy(s2,s1);
            strrev(s2);
            if (strcmp(s1,s2)==0)
               printf("\nGiven strnig is palindrome ");
            else
               printf("\nGiven string is not palindrome ");
            getch();  }
```

ARRAYS

Introduction

What is the use of variable? Generally a variable is used to store the value which may be used for further reference. In a simple variable only one value can be stored at a time. For example to store **5** values, we can use the variables like **a, b, c, d** and **e**. But when the number of values to be stored is more the difficulty will also be more. So, the solution for the above problem is using an array.

*) Array is a collection of elements or data items
*) All the elements must be same data type
*) and they are stored in consecutive memory locations

How to Declare Array variable?

Simple variables are declared as,

```
int a,b,c;   /* Simple variable Declaration */
```

data-type followed by list of variables.

Similarly, an array variable can also be as follows.

```
data type   variable   [ size ] ;
```

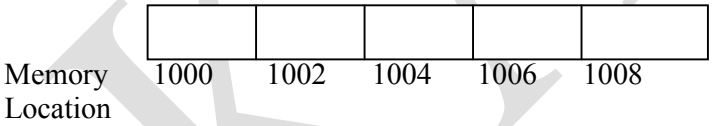
Where - **data type** is the type of data like int, char etc.,
- **variable** is the name of the array variable
- **size** is the maximum no. of elements to be stored in the array and the size must be an integer value.

Example for array declaration is

```
int  a [ 5 ] ;
```

*) **a** is the name of the array variable of type **int**eger *) In the variable **a,** we can store **5** integer values.

*) The memory allocation is as follows which assumes the starting address is **1000.** Each element of the array occupies **two** bytes because of integer data type.



How to refer the values of array variable?

In **C** the first element of array is stored at the location **0**. So the element can be accessed as follows:

- First element is referred as a[0] location 1000
- Second element is referred as a[1] location 1002
- Third element is referred as a[2] location 1004 etc.

*) Here **0,1,2, …** are called as subscript (or) index. So array is also called as **subscripted variable.**

*) The above array is called **single dimensional array**. Because to refer any data we need only one index.

Assigning the data into array:

Like a simple variable assignment, the values can be assigned to the array variable as shown in the example.

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

Here 1st element is stored at a[0]
2nd element is stored at a[1]
3rd element is stored at a[2]
4th element is stored at a[3] and
5th element is stored at a[4]

Following are some of the ways to assign values to array variables.

*) int a[10] = {20,30};
=> Here **10** memory locations are reserved for **a**. But we are using only **2**. So the remaining spaces (eight) are wasted.

*) int a[ ] = {10,20,30};
=> In this declaration the above problem has been solved. The size of array is adjusted automatically depending upon the no. of values assigned.

*) The **default value** of variable is **garbage value**.
*) The following declaration initializes all the values of array to **0**.

```
int a[100] = {0};
```

In this case all the locations are filled by the value **0**.

Relationship between loop and arrays

The elements in an array are referred by using the array name and the index number. For example in the array

```
int a[10];
```

The individual elements are referred by

a[0] , a[1],a[2],a[3] ……. a[9].

If the number of elements is less, we can use the indices to refer the individual elements. But if we have to refer 10s of 100s of elements sequentially what can we do? Can we have 100 statements to refer to the individual elements? Of course, it can be, but requires lot of typing like:

```
printf("%d",a[ 0 ] );
printf("%d",a[ 1 ] );
  . . .
printf("%d",a[ 9 ] );
```

The alternative way is to use a looping in which the index can be varied as shown below.

```
for (i=0;i<9;i++)
    printf("%d",a[ i ] );
```

So, for processing array we can use the looping statements. Compare the processing difficulties with looping statement and without.

```
/* Example for Reading Array and Displaying it */
main( )
{
 int i, a[5];
 printf("\nEnter 5 Elements for array \n");
 for(i=0;i<5;i++)
        scanf("%d",&a[i]);
 printf("\nYour Given values are\n");
 for(i=0;i<5;i++)
        printf("%d\t",a[i]);
}
```

Example: To find the biggest of N numbers, the algorithm is:

1. Read n numbers into the array
2. Assign the first element of array to a variable BIG assuming that it is the biggest one.
3. Compare this element with next element of array.
4. If the next element is bigger, assign this value as the new value of big.
5. If not, keep the existing values as BIG
6. Continue steps 3 to 5 till the last element is compared.

```
/* To find maximum no. from the N values. */
main( )
{
 int i, n, max, a[5];
 printf("\nHow many values :");
 scanf("%d",&n);
 printf("\nEnter %d values \n",n);
 for(i=0;i<n;i++)
        scanf("%d",&a[i]);
 printf("\nYour Given values are\n");
 for(i=0;i<n;i++)
        printf("%d\t",a[i]);
 max=a[0]; /* Assume a[0] is a maximum */
 for(i=1;i<n;i++)
        if ( max<a[i] )
                max=a[i];
 printf("\nMaximum Number : %d ",max);
}
```

Example: To insert new data item in the list.

For example, an array contains

A[1]=10; A[2]=20; A[3]=30; A[4]=40; A[5]=50;

If we want to insert a new value in middle, we need the information like the place to insert and value to be stored.

For example, if we want to insert a new item in the 3rd location, the 3rd location value has to be moved to 4th and 4th moved to 5th etc. Finally the 3rd location will be empty and we can assign the value to that location. The movement of location starts from last. Otherwise the values will be overwritten.

In general **i th** location value will be adjusted to **(i+1) th** location.

```
/* To insert a new item in the specified location */
main( )
{       int i,n,p,value,a[10];
        printf("\nHow many values :");
        scanf("%d",&n);
        printf("\nEnter %d values \n",n);
        for(i=0;i<n;i++)
          scanf("%d",&a[i]);
        printf("\nWhere to        insert a new value :");
        scanf("%d",&p);
        printf("\nValue of that location :");
        scanf("%d",&value);
        printf("\nData Before Inserting\n");
        for(i=0;i<n;i++)
          printf("%d\t",a[i]);
```

```
        for(i=n-1;i>=p;i--)
           a[i+1] = a[i];
             /* Adjusting the ith location to i+1 */
          a[p] = value;
          printf("\nData  After Inserting\n");
          for(i=0;i<=n;i++)
                    printf("%d\t",a[i]);
        }
```

Output:

```
How many values :5
Enter 5  values
10 20 30 40 50
Where to insert a new value :3
Value of that location :100
Data Before Inserting
10      20      30      40      50
Data  After Inserting
10      20      30      100     40      50
```

More Dimensions - Multi Dimensional Array:

The single dimensional array is suitable for simple applications like storing the marks of students in a subject. Because only one information is enough to refer the mark of a particular student.

But to store the marks of students in different subjects, then we need **two-dimensional** array. Because to refer mark of any students, we need one more extra information to know which subject is required.

For example, marks [1][5] means , subject 1 and 5th student.

A very good example for two-dimensional array is a **matrix,** which is having no. of columns and no. of rows. So to refer any element of a matrix we need two information one is row number and another one is column number.

How to Declare Two-dimensional Array?

Similar to single dimensional array and simple variable declaration, the two-dimensional array is declared as

```
Data type   variable  [ size1]  [ size2 ] ;
```

Where - **size 1** refers to no. of rows
- **size 2** refers to no. of columns

So totally we can store **size1 * size2** values in the **variable**.

Example: int a[20][10];

*) Here **a** is the two dimensional array variable and it has 2**0** rows and **10** columns. So totally we can store **200** (20 * 10=200) values in the variable **a**.

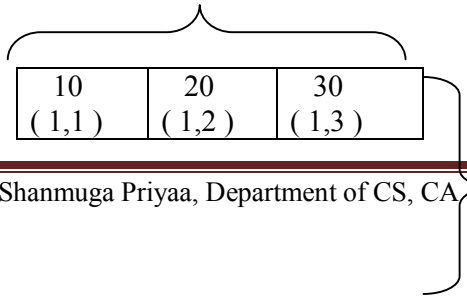
*) To refer any value we have to specify the row no. and column no.

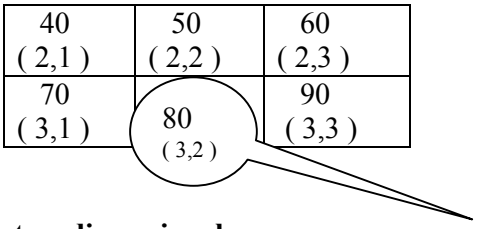
Ex: a[2][5]

=> It refers to the element of 2nd row 5th column

Suppose there is a matrix with 3 rows & 3 Columns, the representation is as follows.

Columns (3 Columns)





How to process in two dimensional array:

The elements are processed by specifying its indexes like which row & which column. For example the above array is accessed as follows.

a [1] [1] => First row's First Column's value
a [1] [2] => First row's Second Column's value
a [1] [3] => First row's Third Column's value
a [2] [1] => Second row's First Column's value
a [2] [2] => Second row's Second Column's value
. . .

Here **9**, nine statements are necessary to process all the elements of array. When the number of dimensions increase, the number of statements will also increase.

Is there any relationship among the index of arrays. Yes. The index is in sequence. In a matrix (rows & Columns) each row contains number of columns. Can we simplify the above statements. Think about the nested loops, in which for every value of first loop second loop will execute N times. Apply this approach here. Assume there are two loops (nested) namely **i** and **j,** for row and columns respectively. So, every value of **i**, **j** will execute N times.

```
for ( i=1 ; i < n; i++ )
   for ( j=1 ; j < n; j++ )
         // Statement of nested loop
```

Here statements are executed **n * n** times. That is for every value of **i**, the **j** will be executed **n** times.

The individual statements are simplified by using nested loop is as like below

```
for ( i=1 ; i < 3 ; i++ )
   for ( j=1 ; j < 3 ; j++ )
         printf( "%d", a [i][j] );
```

Think about the relationship between two-dimensional arrays and the nested looping statements.

Assigning Values

We can assign the values to the variable while declaration of it.

```
Ex :  int  a[2] [3]  = {   {1,2,3},
                           {4,5,6}                    };
```

*) Each row is separated by braces **{ }** and each element by commas.
*) Here two rows and three columns.
*) First row contains the values 1,2,3 and second row contains the values 4,5,6.

Example to read and display of two-dimensional array (Matrix).

```
/* Example for reading and displaying matrix */
  main( )
  {
   int i,j,r,c,a[5][5];  /*Declared as 5 x 5 matrix*/

   printf("\nNumber of rows : ");
   scanf("%d",&r);
```

```
            printf("\nNumber of column : ");
           scanf("%d",&c);
            printf("\nEnter %d x %d matrix\n",r,c);
            for(i=0;i<r;i++)
              for(j=0;j<c;j++)
                    scanf("%d",&a[i][j]);
            printf("\nYour matrix is \n");
            for(i=0;i<r;i++)
                    {
                     for(j=0;j<c;j++)
                     printf("%d\t",a[i][j]);
           printf("\n");
                     }
     }
```

Example:

Multiplication of the matrix is different from the addition of two matrices. In multiplication, no. of columns in the first matrix and no. of rows in the second matrix must be equal. Simple example for the matrix multiplication is

```
 /* Matrix Multiplication. with equal rows & colms  */
  main( )
  {
    int i,j,k,row,col,a[5][5],b[5][5],c[5][5];
         printf("\nNumber of rows    : ");
    scanf("%d",&row);
    printf("\nNumber of Columns : ");
    scanf("%d",&col);
    printf("\nEnter first matrix values\n");
          for(i=0;i<row;i++)
                  for(j=0;j<col;j++)
                          scanf("%d",&a[i][j]);
    printf("\nEnter Second matrix values\n");
         for(i=0;i<row;i++)
                  for(j=0;j<col;j++)
                          scanf("%d",&b[i][j]);
    for(i=0;i<row;i++)
                  for(j=0;j<col;j++)
                  {
                   c[i][j] = 0;
                   for(k=0;k<col;k++)
                     c[i][j] = c[i][j] + a[i][k] * b[k][j];
                  }
          printf("\nResult matrix\n");
          for(i=0;i<row;i++)
           {
            for(j=0;j<col;j++)
                    printf("\t%d",c[i][j]);
                  printf("\n");
           }
  }
```

Output:

```
Number of rows    : 2
Number of Columns : 2
Enter first matrix values
1 2 3 4
Enter Second matrix values
1 2 3 4
Result matrix
 7      10
 15     22
```

Example: To transpose of a given matrix. (Columns are posted to rows and rows are posted to columns called transpose of a matrix).

```
/* Transpose of Matrix */
main( )
{
 int i, j, row, col, a[5][5],b[5][5];
 printf("\nNumber of rows    : ");
 scanf("%d",&row);
 printf("\nNumber of Columns : ");
 scanf("%d",&col);
 printf("\nEnter the matrix values\n");

 for(i=0;i<row;i++)
        for(j=0;j<col;j++)
                scanf("%d",&a[i][j]);
 printf("\nTransposed matrix is \n");
 for(i=0;i<row;i++)
   {
    for(j=0;j<col;j++)
        {
         b[i][j] = a[j][i];
         printf("%5d ",b[i][j]);
        }
        printf("\n");
  }
 }
```

Output:

```
Number of rows    : 2
Number of Columns : 2
Enter the matrix values
1 2 3 4
Transposed matrix is
 1     3
 2     4
```

Possible Questions

Part-A (1 mark - Online Examination)

Part-B (2marks)

1. What are void functions?
2. Write notes on i) Call by value ii) Call by reference of functions.
3. Give the difference between "call by value" and "call by reference" in functions.
4. Write the syntax to define a multidimensional array.
5. What is a function? How will you define a function?
6. What is array?

Part-C (6marks)

1. Explain in detail about String functions with syntax and example.
2. What are void and inline functions? Explain in detail with syntax and example.
3. Write in detail about functions that pass variable number of arguments. Explain with syntax and example.
4. What is an array? Explain is detail the various types of arrays with example.
5. Explain the following (a) Function (b) return value (c) arguments.
6. How will you declare and initialize a two dimensional array? Write a C program to perform matrix addition.
7. What are functions? Explain the various categories of functions with syntax and example.
8. How will you declare and initialize a one dimensional array? Give an example program to assign marks of a student in an array.

KAHE

KARPAGAM ACADEMY OF HIGHER EDUCATION					
PART - A (ONLINE EXAMINATION)					
MULTIPLE CHOICE QUESTIONS (Each question carries one mark)					
SUBJECT: PROGRAMMING FUNDAMENTALS USING C/C++					
QUESTIONS	OPTION1	OPTION2	OPTION3	OPTION4	**ANSWER**
____ function joins two strings together	strcat	strcmp	strcpy	strlen	strcat
____ function compares two strings identified by the arguments	strcat	strcmp	strcpy	strlen	strcmp
strcmp function returns the value_____ if the arguments are equal	zero	one	two	three	zero
_____ function assigns the contents of one string to another	strcat	strcmp	strcpy	strlen	strcpy
._____ function counts and returns the number of characters in a string	strcat	strcmp	strcpy	strlen	strlen
Individual values in array is referred as _____	subscript	elements	subelements	pointers	elements
Any subscript between _____ are valid for an array of fifty elements	**0-49**	**0-50**	**0-47**	**0-51**	0-49
Value in a matrix can be represented by _____ subscript	**1**	**3**	**2**	**4**	2
Arrays that do not have their dimensions explicitly specified are called_____	**unsized arrays**	**undimensional arrays**	**initialized arrays**	**to size arrays**	unsized arrays
_____ statement is necessary only when the function is returning some data	continue	return	exit	break	return
The function _____ is used to check whether the argument is lower case alphabet	islower	isupper	tolower	toupper	islower
The function _____converts the lower case argument into an upper case alphabet	islower	isupper	tolower	toupper	toupper
The function _____converts the upper case argument into an lower case alphabet	islower	isupper	tolower	toupper	tolower

_____ character function is used to check white space character	isspace	ispunct	iswhite	isalpha	isspace
_____ character function is used to check alphabetic character	isspace	ispunct	iswhite	isalpha	isalpha
Character test functions return the value _____ if the condition is true	1	0	11	111	1
printf belongs to the category ____ function	**user defined**	**library**	**subroutine**	**preprocessor**	library
_____ is self contained block of code that performs a particular task	function	instruction	program	process	function
_____ statement is the mechanism for returning value to the calling function	return	continue	break	goto	return
A function can return ____ value per call	one	zero	two	multiple	one
_____ is a special case where a function calls itself	recursion	subroutine	structure	union	recursion
C supports _____ storage classes	one	two	three	four	four
_____ static variables are those which are declared within particular function	external	internal	automatic	register	internal
____ variables are declared outside function	external	internal	automatic	register	external
_____ variables are created when the function is called and destroyed automatically when the function is exited	external	internal	automatic	register	automatic
Automatic variables are _____ to the function in which they are declared	local	global	static	protected	local
Automatic variables are also referred to as _____ variables	internal	external	local	static	internal
Automatic variables can also be defined within set of braces known as ____	code	blocks	scope	process	blocks
Variables that are both alive and active throughout the entire program are known as	internal	external	local	static	external

External variables are also known as _____ variables	internal	external	local	global	global
_____ variables can be accessed by any function in the program	internal	register	local	global	global
_____ declaration does not allocate storage space for variables	intern	extern	static	register	extern
_____ variables are those which can retain values between function calls	static	extern	global	register	static
Accessing register variables is much faster than _____ access	program	memory	instruction	comment	memory
All local variables of a function are destroyed when _____	function called	program starts execution	function ends	program ends	function ends
A static variable is initialized once, when the program is _____	compiled	executed	developed	closed	compiled
_____ variables have local scope but its value persists until end of the program	intern	extern	static	register	static
_____ functions has to be developed by the user at the time of developing a program	user defined	built in	subroutines	structure	user defined
_____ functions are not required to be written by users	user defined	built in	subroutines	structure	built in
_____ header file should be decalared to call clrscr function	stdio.h	stdlib.h	conio.h	math.h	conio.h
_____ header file should be decared to call input and output function	stdio.h	stdlib.h	conio.h	math.h	stdio.h
Header file stdio.h calls _____ function	input/output	math	character	sqrt	input/output
Variable initialized once during compilation are _____	static	register	local	global	static
Global variables are also known as _____	static	register	local	external	external
Internal variables are also called _____	static	register	auto	global	auto

_____ function used to find the absolute value of a floating point number	ceil()	int()	fabs()	sqrt()	fabs()
C will automatically convert _____ variables into non register variables	internal	external	register	auto	register
Recursion is a process of function calling _____	another function	itself	subroutine	structure	itself
_____ returns zero or one value per call	function	structure	arrays	pointers	function
_____ function is used to convert characters into ASCII (integer) values	toupper	atoi	tolower	itoa	atoi
Individual members of structures compared using _____ operator	arithemetic	logical	bitwise	comma	logical
In function declaration if the return type is not specified , it returns _____ by default	integer	character	float	long	integer
Functions are _____ by default	extern	intern	register	auto	extern
_____ external variables can be accessed by other files	simple	extern	intern	auto	simple
Functions should be declared as _____ if it is to be accessible only to the functions in the files	static	extern	intern	auto	static
Variables enclosed within function paranthesis are called_____	arguments	variables	data	elements	arguments
Modular structure of C language enables the program to be split into several modules called	structure	union	integers	function	function
The actual and formal arguments of functions must match in ____	number of arguments	type of data	order in which they appear	all the above	all the above
Arguments passed in the function call statement are called _____	actual arguments	formal arguments	dummy parameters	temporary variables	actual arguments
Functions receives the values passed by the calling function and stores in_____	actual arguments	constants	dummy parameters	temporary variables	actual arguments

UNIT III

Syllabus

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

Structures

Introduction - What is structure & Why?

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:

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

Example:

```
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.

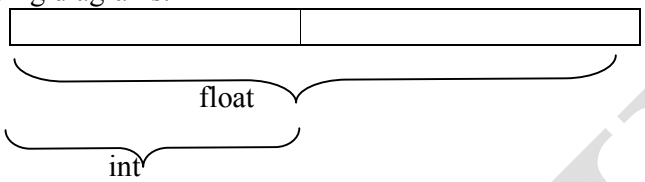
```
/* Example for the union variable */
main( )
{union
```

```
                { 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

```
#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.

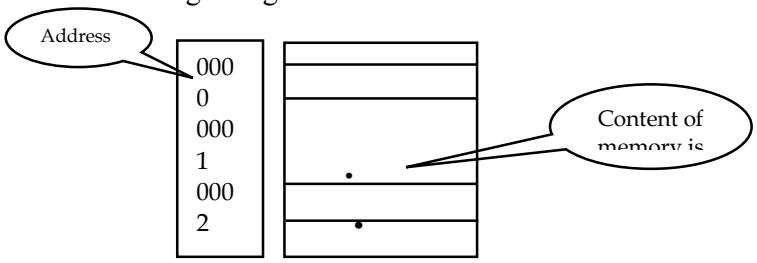
What is pointer?

- 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



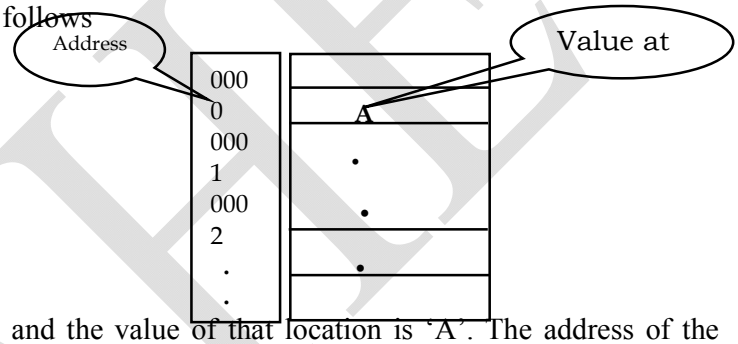
allotted for them. Memory is divided into small pieces to keep small data called byte (8 Bits). Our program and data will be stored in somewhere in the memory where the free area is available. The variables are the names used for reference but everything internally referred by the memory address. Let 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.

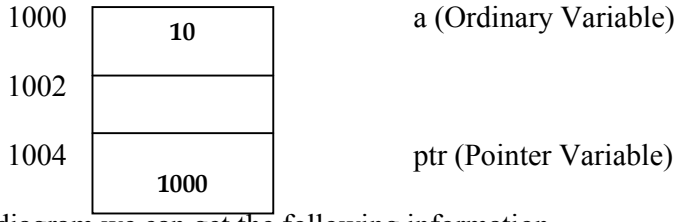
1000	**10**	a (Ordinary Variable)
1002		
1004		ptr (Pointer Variable)

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



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;
```

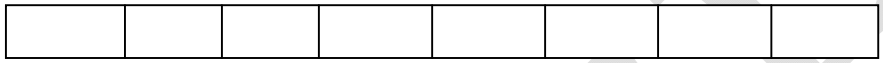
```
    int *ptr;
    ptr=&a;
    printf("\nOld Value of a = %d ",a);
    *ptr=100;
    printf("\nNew Value of a = %d ",a);
}
```

```
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

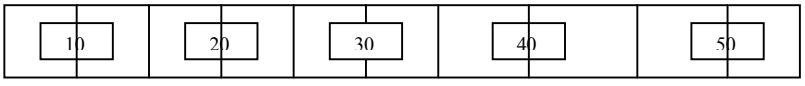
Note:

- ⇨ 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 };



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

Standard deviation = $\sqrt{\text{variance}}$

Where

Variance = $1/n \sum (Xi - Mean)^2$ and i= 1 to n

Mean = $1/n \sum Xi$

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] ;

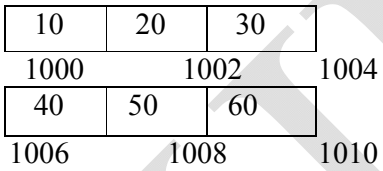
This declaration tells

- ⇨ 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 declaration of structure pointer is as follows.

```
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);
  }
```

STORAGE CLASSES

The variable can be declared with additional functionality by using storage classes. When we add these features, the variable may be with different qualifications like scope, default value etc, are different to compare with ordinary variable declaration.

The variable declaration with storage class is

StorageClass Datatype Variable(s);

There are four storage classes available in C and they are

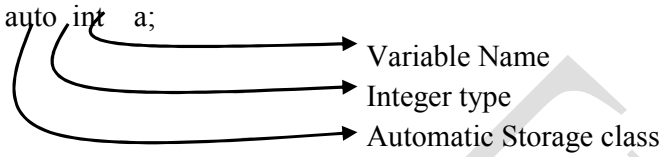
1. Automatic variable
2. Static variable

3. External variable
4. Register variable

Automatic variable:

- ⇨ The variable of automatic storage class can be declared using the key word **auto**.
- ⇨ In default all the variables are **auto**matic variable. So, it is optional.
- ⇨ Automatic variable contains a **garbage value** by default.
- ⇨ **Life** of the variable is **local** or inside the **block**

The variable declaration with **auto**matic storage class is as follows.



Automatic is optional , so the above declaration can be replaced by the traditional statement int a; /* Default automatic variable */

Static storage class:

Sometime we are in a position to retain the last value of the evaluation in a program or initialization should be only once for many function calls, to obtain this feature, **static** storage class is used, some additional information about it

- ⇨ It is used to keep the values as static
- ⇨ Default value of static variable is **0** (Zero)
- ⇨ Life is until the program termination
- ⇨ Initialization of a variable is only once
- ⇨ Scope is local only

When the variable is declared as static, its default value is assigned as 0 (Zero). Look the following example and you may clear the above.

```
main ( )
{
 static  int  a ;
 printf("Value of a = %d ",a);
}
    Value of a = 0
```

Another characteristic is one time initialization, this is illustrated in the following program.

```
/* Example for static storage class */
main( )
{    int i;
    for(i=1;i<=3;i++)
 {   static int a=10;
   printf("\na = %d ",a);
   a++;
 }      }
```

Here the variable **a** is declared inside the **for** block. So, for every execution there is a variable declaration. But here with additional keyword static. So the initialization is only once not every execution. You check the same program without **static** storage class. For every execution the result is same.

Another characteristic is scope of the variable. The scope of the variable is only inside the block not everywhere.

```
/* Example for static with block */
main( )
{   int a=10;
   printf("\nValue of  a = %d ",a);
   {       static int a=100;
      printf("\nValue of  a = %d ",a);
    }
    printf("\nValue of  a = %d ",a);
}
```

```
Value of  a = 10
Value of  a = 100
Value of  a = 10
```

Here the scope is local, so while exiting from the block the variable is not taken into account and removed from the memory.

One more characteristic is local. That is the variable could not be accessed outside of the function.

```
/* Using static  */
main ( )
{
  int i;
  for(i=1;i<=5;i++)
        test( ); }
test( )
{   static int a=5;
  printf("\na = %d ",a);
  a++;
 }
```

```
  a = 5
  a = 6
  a = 7
```

```
/* Without using static  */
 main ( )
 {
   int i;
   for(i=1;i<=5;i++)
         test( );  }
 test( )
 {    int a=5;
    printf("\na = %d ",a);
     a++;
 }
```

```
      a = 5
      a = 5
    a = 5
```

External variable:

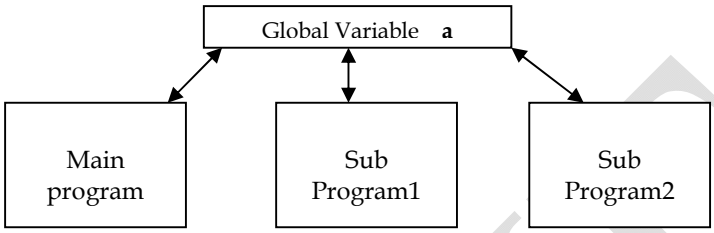
We have discussed so many programs and all of them are based on the local variable. Suppose any variable is going to be referred all the places of the program, we can declare that variable as **extern**al variable. This type of declaration is also called as **global variable** declaration and can be declared using the keyword **extern.**

The external variable should be declared outside of the main( ) function or before the main( ) function. If it is declared outside of the main( ) there is no need of the keyword **extern**. The following is a piece of code with external variable declaration.

```
    int  a = 10;    /*  Global declaration */
main ( )
    {
```

```
    // Statements of main function
}
void  test( )
{
   // Statements of user defined function
}
```

The visibility of the global variable **a** is illustrated in the following diagram. Here the variable **a** is visible not only to main() function but also all the sub-programs.



The usage of external variable is explained in the following program.

```
/* External storage Class  */
extern int a=10;     /* Global variable */
main( )
{printf("\nIn main a = %d ",a);
        test( );
}
void test()
{    printf("\nIn function  a = %d ",a);
}
```

```
In main a =  10
In function  a =  10
```

From the result we can conclude that the variable **a** can be accessed any where in the program. One more program, which illustrates that any changes will affect the global variable value.

```
/* Example for external variable */
int a=10;
main( )
{    printf("\nBefore Call  a  = %d ",a);
     a=a+10;
     test( );
     printf("\nAfter  Call  a  =  %d ",a);
}
void  test( )
{printf("\nValue of  a in function  = %d ",a);
a=a+10; }
```

```
Before Call  a  = 10
Value of  a in function  = 20
After  Call  a  =  30
```

Register storage Class

What is register? Register is a small area, which is used to store the data temporarily while doing calculation. Few important registers are like AX,BX, Count Register, Status register. Main uses of these registers are to reduce the time to perform the calculations. In general the value is stored in memory (RAM) and it will move to the register for the process. But it takes time if the process is very often. In this situation we can keep the values in the register itself. So the memory movement will be reduced.

register int i;

Here the value of **i** is stored in the register instead of memory. If it is going to be used frequently, like in the looping statement, it is very useful.

```
/* Example for register storage class */
main( )
{       register int i;
        for( i = 1;i < =100; i++)
        printf("\n%d ",i);}
```

Note : Generally Register size is 16 bit. So there is no possibility to store the float or double value.

KAHE

Review Questions

1. Define structure? Discuss the necessity of structure in a program
2. How to declare structure? Example
3. How the members of structure are accessed?
4. Write short notes on array of structure.
5. Write short notes on nested structure.
6. Explain about how the structure variables are passed to the function.
7. Explain with example how the memory allocation is made for structure.
8. What is the advantage of union?
9. Differentiate structure and union with an example
10. Write short notes on typedef.
11. Write short notes on enumerated type
12. What is the need of function in a program?
13. What are the two general types of function?
14. Write some example for library function
15. Write the general format of function definition
16. What is parameter?
17. Write short notes on return statement
18. Discuss the various types of function with necessary examples
19. How can we pass the whole array to the function?
20. What is the use of void type?
21. What is the Default return type of the function ?
22. Define recursive function. Discuss with example
23. Write a program to find sum of N numbers using recursive function
24. What is calling convention?

Review Questions

1. What is the need of pointers in a program?
2. What is the disadvantage of function?
3. How to declare pointer variables?
4. What are the operators associated with pointer variable?
5. How to obtain the address of a variable?
6. Write short notes on pointer arithmetic
7. Write short notes on pointers and function
8. Explain the role of pointers in string with an example
9. Call by value and Call be reference – Explain
10. How to pass the whole array to the function?
11. Write short notes on pointers and two – dimensional array
12. Prove that how the array elements are stored in consecutive memory locations
13. How do we use array of pointers? Explain
14. Can we call a function using pointers? If so how?
15. What is the need of Dynamic memory allocation?
16. How to allocate memory dynamically>
17. Why we have to release the memory and how?

KARPAGAM ACADEMY OF HIGHER EDUCATION					
PART - A (ONLINE EXAMINATION)					
MULTIPLE CHOICE QUESTIONS (Each question carries one mark)					
SUBJECT: PROGRAMMING FUNDAMENTALS USING C/C++					
QUESTIONS	OPTION1	OPTION2	OPTION3	OPTION4	**ANSWER**
Process of calling a function using pointers to pass address of variable is known as	call by value	call by reference	call by method	call by address	call by reference
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
In pointers when function is called _____ are passed as actual arguments	values	addresses	operators	objects	addresses
In pointers _____ is called arrow operator	>	<	*	&	>
_____ method is for packaging data of different data type	structure	union	pointer	function	structure
_____ keyword is used to declare structure	bit	struct	union	array	struct
Fields within structure are called _____	data	variable	member	subelements	member
Linking member and a variable in structure is established using	*	&	dot	>	dot
Structure must be declared _____ if it is to be initialized inside function	intern	extern	static	register	static
_____ operator is used to find the number of bytes occupied by a structure	size	atoll	sizeof	strlen	sizeof
_____ is convenient tool for handling group of logically related data items	union	structure	bitfields	array	structure
All members of _____ use the same storage location in memory	union	structure	bitfields	array	union
If p is an integer pointer with value 2800, then after the operation p=p+1 ,the value of	2801	2802	2803	2804	2802
In incrementing pointer , value is increased by the length of the data type it	increment factor	scale factor	size factor	keywords	scale factor
The ____ address is the location of the first element of the array	memory	base	starting	end	base
Pointers can be compared using _____ operators	logical	relational	arithmetic	conditional	relational
Pointer cannot be used in_____	addition	subtraction	multiplication & division	modulo	multiplication & division
A pointer contains _____ value until it is initialized	initial	address	garbage	null	garbage
In pointer scale factor for character data is _____ byte	1	2	4	8	1
_____ operator is used to compare all members of structure	=	==	+	**	==
___ operator is used to assign value of one structure to another variable	=	==	!=	assign	=

On comparing two structure variables of the same structure using = = operator	0	3	1	2	1
On comparing two structure variables of the same structure using != operator	0	1	TRUE	FALSE	0
In pointer * operator is known as _____operator	direction	indirection	arrow	dot	indirection
_____ are constructed or derived data types	structure	integers	string	long double	structure
_____ is a collection of logically related data items of different data type	array	structure	string	float	structure
The name of the structure is called_____	structure tag	structure element	field name	item name	structure tag
The structure template must end with a _____	_ underscore	; semicolon	. Period	: colon	; semicolon
Members of one structure variable can be copied to that of another variable of the	assignment operator (=)	newline operator (\n)	pointer operator (*)	member operator (.)	assignment operator (=)
_____ is called the member operator	* asterisks	; semicolon	. Period	: colon	. Period
Each member of a structure must be declared independently for its	size and name	name and type	height and length	scope and life	name and type
Convenient way of representing the details of all students in a class is	array of structures	two dimensional array	array of names	structure with arrays	array of structures
The individual members of a structure cannot be initialized inside the structure template.	true always	can be initialized if it is static	false always	can be initialized for arrays of	true always
Two structure variables are of same structure. Which of the following operations	assign one variable to another	compare one variable with other	add/multiply one variable with other	all the above	add/multiply one variable with other
Unions have the same syntax as the ____	structures	arrays	pointers	functions	structures
Unions are declared using the keyword_____	struct	union	define	declare	union
_____ contains many members of different data types but can handle only	structure	union	array	strings	union

C language does not permit _____ of structures	partial initialization	initialization of individual elements	initialization of entire variable	assign values using = operator	initialization of individual elements
Each memory location has a number associated with it called	bit	pointer	byte	address	address
Data type of a pointer is _____	signed character or integers	unsigned long integers	same as the data type of variable to	double precision real numbers	same as the data type of variable to which it points
_____ is called the address operator	&	*	%	~	&
Memory addresses are _____ numbers	long integer	unsigned integer	floating point	octal	unsigned integer
_____ reduces length and complexity of program	structure	union	pointers	integers	pointers
What does the following segment of code prints : int x,*p; p=&x; printf("%d",*p);	address of variable x	any integer value	value stored in variable x	address stored in pointer p	value stored in variable x
In 16 bit machine, a pointer that is pointed to float type of data will have scale factor	2	4	8	16	4
Nesting of structures means _____	array of structures	structure with arrays	structure within structure	structure with functions	structure within structure
_____ enables us to access a variable that is defined outside the function	structures	unions	pointers	arrays	pointers
Use of _____ arrays to character strings saves storage space in memory	character	float	integer	pointer	pointer
p is a pointer to 2 D array, then _____ points to the first element in i^{th} row	p+i	*(p+i)	*p + i	p+*i	*(p+i)
*p[3] declares p as	pointer to an array of three elements	an array of 3 pointers	pointer to the number 3	function of 3	an array of 3 pointers
Making a pointer point to another pointer is called	nested indirection	recursive indirection	multiple indirection	array indirection	multiple indirection
What does the following segment of code prints : int x,*p; p=&x; printf("%u",*p);	address of variable x	address of pointer p	value stored in variable x	address stored in pointer p	value stored in variable x
Pointers can be made to point to _____	functions	arrays	structures	all the above	all the above
During structure initialization, the uninitialized member will be	\0	000u	0x	0	0
During structure initialization, the unitialized member will be assigned a	\0 '	000u	0x	'	\0 '
Initializing only first few members of a structure and leaving other members	primary	secondary	partial	first	partial
Which of the following is not true	A pointer variable can be assigned the	It can be assigned the value of	It can be prefixed with an increment	It can be multiplied by a constant	It can be multiplied by a constant
When an array x is assigned to a pointer variable it points to	the base address of x	address of x[1]	the last address of x	address of x[0] + 1	the base address of x
Which of the following is the correct output for the program given below?	llo	hello	ello	h	hello
Which of the following is not correct	char str[5]="good";	char *str = "good";	char *str; str="good";	char str[5]; str="good";	char str[5]; str="good";
Which of the following is not correct	int *p, x=5; p= &x;	int *p; p=&5;	int *p,x,y; x=y+5; p=&x;	int *p; *p=5;	int *p; p=&5;

Class: I B.Sc IT **Course Name: Programming Fundamentals Using C/C++**
Course Code: 18ITU101 **Unit: IV (Memory Allocation, File I/O)** **Batch 2018-2021**

UNIT-IV

Syllabus

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.

Files

Introduction

The program upon execution will require some data as input to be processed to yield the results. As the data are stored temporarily, the data has to be reentered to see the results. Imagine, if the volume of the data is high like 1000 records, is it possible for us to reenter the voluminous amount of data each time. This is the major draw back of previous programs. This can be overcome by using files.

The file-based programs are used to store the data permanently and can be referred in the future. **FILE** is a predefined structure, which maintains all the information about the files we create. Information such as the location of buffer, the pointer to the character in the file, the end of file and mode of the file etc. We may perform the following operations in a file

1. Read data from the file
2. Write data into the file

The operation of reading and writing may be performed either character by character or word by word or line by line or record by record.

How to declare FILE type variable?

We already know **FILE** as a predefined structure. The file pointer is declared as follows.

```
FILE * file-pointer;
```

Where **file-pointer** is a pointer variable, which points to the first character of an opened file.

Example: FILE *fp; /* fp is a file pointer */

Here **fp** is a **file-pointer,** which points to the first character of the file (Starting address of the file). If we know the starting address then we can access all the data of file using the features of pointers by adjusting into its next locations.

How to Open / Close the file?

The purpose of using a file may be to read or write data. To do any of the two functions first we must open the file. The file can be opened by using the library function **fopen( )** and its format is

Where **filename** is the name of the file to be opened and **mode** is the purpose of opening the file like read / write / append.

```
FILE *fp = fopen(filename, mode);
```

If the file exists, it is opened and the **fopen( )** function **returns the starting address** of t he file. If **fail** on opening the result is **NULL**.

For example

fopen("stu.dat","r");

=>"**stu.dat"** is the name of the file and "**r"** is mode of opening the file ("**r"** indicates the read mode). If success, it returns the starting address of **stu.dat** file.

The file may be opened in any one of the following modes:-

Mode	Description
w	Create a new file for writing
r	Open the available file for reading
a	Open the file for appending
w+	Create the new file for both operations (read & write)
r+	Open the available file for both operations (read & write)
a+	Open the available file for both (read & write) operation with append

- If an already existing file is opened in writing mode the contents of it will be lost and a new file is created with the same name.
- If an already existing file is opened in append mode the contents of will not be lost and a new information can be added to the existing data.
- If you are going to handle binary files then you will use below mentioned access modes instead of the above mentioned:

"rb", "wb", "ab", "rb+", "r+b", "wb+", "w+b", "ab+", "a+b"

Examples with different type of opening modes are given below, assume the result is assigned to a **FILE** pointer variable.

1. fopen("stu.dat","r")
   - **stu.dat** is opened for reading the data and file must exist.
   - If no such file exists, this function returns **NULL** value.
2. fopen("stu.dat","w")
   - The file **stu.dat** is created for writing the data.
   - Every time a new file will be created with the same name.
   - Data will be overwritten
3. fopen("stu.dat","a")
   - The file **stu.dat** is opened for appending data at the end of file.
   - If no such file , new file is created for writing.
   - New data will be added to the end of already existing data.
4. fopen("stu.dat","r+")
   - The file is opened for both purposes (Reading and Writing).
5. fopen("stu.dat","w+")
   - The file is created for both purposes (Reading and Writing).
6. fopen("stu.dat","a+")

- The file is opened for both purposes. If the file is not existing, new file is created.

The complete relationship between the file pointer and assignment of file to it is illustrated below:

```
FILE *fp;
fp = fopen("test.dat","r");      or
FILE *fp= fopen("test.dat","r");
```

In this declaration **fp** is a file pointer and it points to the first character of the file **test.dat**. Suppose the file does not exist, the **fopen( )** returns **NULL** value. You can use the following program to check the availability of the file.

```
/* Example for testing the availability of the file */
main( )
{    FILE  *fp;
```

```
        fp = fopen("test.dat","r");
        if (fp= = NULL)
          {    printf("\nNo such file in the directory.");
           exit(1);    /*To exit from the running program  */
      }
     fclose(fp);     }
```

fclose( ) - To close the file

The file is opened for doing the process of reading or writing data. After completion of the process the file has to be closed. If the file is not properly closed the file pointer may be somewhere in the file. If the file is closed the file pointer will successfully start from the first character, the next time.

The function **fclose ( )** is used to close the opened file and **fcloseall()** function is used to close

```
fclose (file-pointer);
fcloseall ( );
```

all the opened file. The general format of these functions is as follows. Example:

1. fclose(fp);- It will close only the specified file.
2. fcloseall( ) - It will close all the opened files.

Reading / Writing character in a file

In this section we are going to discuss about, how to read the characters or write the character into a file. The library functions **fgetc( ), fputc( ), getc( ), putc( )** are used for the above said operations. The following discussions will give an idea about these functions.

getc( ) - To read a character from the file and adjust the file pointer to the next character of the file automatically.
fgetc( ) - to read a character from a file.
putc( ) - to write a character into the file.
fputc( ) - to write a character into the file.

The general format of the getc( ) and fgetc( ) function is as follows.

```
char    getc( file-pointer);
char    fgetc( file-pointer);
```

These functions reads character one by one from the file and the file pointer will be automatically adjusted to the next location in the file. By this way we can read all the characters from the file, sequentially. The characters are fetched from the file and it can be displayed as like below.

```
ch = fgetc(fp);   /* Getting character from file */
putch(ch);              /* Displaying it in the screen */
```

Note: *File pointer will be automatically adjusted to the next location*

feof( ) - To check end of file

During the processing of the file, the end of file can be realized by the value the file pointer returns. The file pointer returns **0** if the end of file has been reached.

The end of file can be checked by using the library function **feof( )**. This function returns **TRUE** if the file reaches the end, otherwise **FALSE**. Using **feof( )** the process to be carried out in the file can be continued. For example, the statement

```
FILE  *fp=fopen("test.dat","r");
result =  feof(fp);
```

If the file reaches the end, the value in the **result** is **TRUE** otherwise the value in the **result** is **FALSE**. We can use a while loop for reading no. of characters as follows.

```
while( !feof( fp ) )
{
  /* Statements   */
```

```
        }
```

Here the statement part will be executed until the end of file. Using a predefined constant, **EOF** can also check the end of character in the file. The following program will give an idea about how to read and check the end of file character.

```
/* Reading the characters from the file */
#include <stdio.h>
main( )
{   FILE *fp;
  char ch;
   fp=fopen("test.txt","r");
  if (fp= =NULL)
  {      printf("\nNo such file this name");
     exit(0);   }
 while (!feof(fp))
 {       ch = getc(fp);
         putch(ch);
 } }
```

In this program the file **test.txt** has been opened for reading the data. Using the function **getc(fp)** the characters are read, and displayed in the screen using **putch( )**. The function **feof( )** is used to check the occurrence of end of file. The process will be terminated when end of file is reached.
The general format of the **putc( )** and **fputc( )** is as follows.

```
putc  ( character , file-pointer);
fputc ( character , file-pointer);
```

where the **character** denotes the character to be written and **file-pointer** indicating the file which is receiving the character. These functions are used to write characters into the specified file.
For example

```
    fputc('a', fp);
        /* character 'a' is written in the file */
```

The following example illustrates the **putc( )** function.

```
/* Creating New file by reading characters */
#include <stdio.h>
main( )
{ FILE *fp;
 char ch;
 fp=fopen("test.txt","w");
 if (fp= =NULL)
   {  printf("\nNo such file");
      exit(0);        }
  while((ch=getch( )) != 'z' )
  {   putc(ch,fp);
      putch(ch);            }
fclose(fp);     }
```

While executing this program the user can type any no. of characters and all the characters will be stored in the file **test.txt**. The process will be continued until the key **z is** pressed. Instead of **z** any other character can also be used to denote the end.
For example the user can give the following text, and the termination is with the character **z.**

No gain without pain z

Using the functions **getc( )** and **putc( ),** content of one file can be copied into another file as in the following example program.

```
/* Copy the content of one file to another  */
#include <stdio.h>
main( )
{ FILE *fps,*fpd;    /* source and destination */
 char ch, sfile[15], dfile[15];
 clrscr( );
 printf("\nSource file       : ");
 scanf("%s",sfile);
 printf("\nDestination file : ");
 scanf("%s",dfile);
 fps = fopen(sfile,"r"); /*source file should be opened for read  */
 fpd = fopen(dfile,"w"); /* This file should be opened for write  */
 if  (fps= =NULL)
 {    printf("\nNo source file in the directory");
      exit(0);      }
 while(!feof(fps))
 {       ch = getc(fps);
         putc(ch,fpd);
 }
 fcloseall( );    /* Closes  all the opened files */
}
```

Output:

```
Source  file      :  ek.c
Destination  file  :  mahi.c
```

After the execution, the content of **ek.c** is copied into the file **mahi.c**

Read / Write - line of characters

We have seen and discussed about how to read and write characters in the file. Let us now see how to read or write a line or string in the file. The function **fputs( )**, **fgets( )** are used for this purpose. The format of the functions is as follows.

```
fputs( string , file-pointer);
fgets(string-var,  size , file-pointer );
```

string is the characters to be written in to the file using file-pointer.
string-var is the variable denoting the string
size is the no. of characters to be accommodated in the variable.

Mostly these functions are used for line by line process and few examples are

1. fputs("Abdul Kalam",fp);
   - Here the string **"Abdul Kalam"** is written in to the file **fp**.
2. fgets(str,80,fp);
   - Here **80** characters are read from the file **fp** and copied into the string variable **str**.

The following program is used to count the no. of lines in the specified file.

```
/*  To count no. of lines from the file  */
```

```
#include <stdio.h>
main( )
{  FILE *fp;
  int c=0;
  char ch, str[80];
  fp = fopen("test.txt","r");
  if (fp= =NULL)
   { printf("\nNo source file in the directory");
     exit(0);
   }
   while(!feof(fp))
   {      fgets(str,80,fp);
            c++;
            puts(str);
   }
   printf("\nNo. of lines :  %d ", c);
   fclose(fp);
}
```

In the above program, for every execution, **80** characters will be picked up from the file **"test.txt"**, and the counter variable **c** will be incremented.

Formatted Input / Output statement in File:

The readers are familiar with the use of formatted input and output functions **scanf( )** and **printf( )**, as we have discussed them so far in many programs. It also supports its usage in files, the functions **fscanf( )** and **fprintf( )** are utilized for the same. The format is as follows.

```
fprintf(file-pointer , format string, variables);
fscanf(file-pointer , format string, variables);
```

The function **fprintf( )** accepts a sequence of arguments (variables) and applies it to the format specifier, which is in the format string and redirect the formatted output to the specified file. There must be the same number of format specifiers and arguments. The **printf( )** function writes the data on the screen, so do the **fprintf( )** writes the data into the file.

The function **fscanf( )** reads the values from the file, and each field is formatted according to the format specifier passed to **fscanf( )** in the format string. Finally **fscanf( )** stores the formatted input at an address passed to it as an argument following the format. The **scanf( )** reads data from the keyboard so do **fscanf( )** reads data from the file stream.

```
char name[ ] = "Bill Gates";
fprintf (fp,"Welcome to %s ",name);
```

Here the format string has been replaced with its value **"Bill Gates"** and is written in to the file.

```
fprintf(fp,"name = %s  a= %f  = %d ",name, a, b);
```

- Here format string has been combined with different data type and as usual they are written in the file **fp**.

```
fscanf(fp,"%d",&n);
```

- Here the data is taken from the file **fp** and copied into the integer variable **n.**
- While reading the data we should be careful about the data type.
- If the format string does not exactly match, then wrong data will be retrieved from the file.

The program below illustrates the preparation of electricity bill and the utility of **fprintf( )** can be visualized.

```
/* To prepare E.B.  Bill for the consumer  */
main( )
{     FILE *fp;
  int serno;
  char name[50];
  float amt, pr, cr, units;
        fp = fopen("eb.dat","w");
        printf("\nConsumer  Name : ");
        gets(name);
        printf("\nConsumer  N0.  : ");
        scanf("%d",&serno);
        printf("\nPrevious reading :");
        scanf("%f",&pr);
        printf("\nCurrent  reading : ");
        scanf("%f",&cr);
        units = cr - pr;
        amt = units * 1.25;  /*Cost per  unit is  assumed  1.25  */
       fprintf(fp,"\t\tELECTRICITY  BLL\n");
        fprintf(fp,"\nName  : %s \tService No. : %d \n",name,serno);
        fprintf(fp,"\n-------------------------------------");
        fprintf(fp,"\nPrevious  Current   Units     Amount");
        fprintf(fp,"\nReading   Reading   Consumed    Rs. ");
        fprintf(fp,"\n-------------------------------------");
        fprintf(fp,"\n%5.2f\t   %5.2f     %5.2f    %5.2f",pr ,cr,units,amt);
        fprintf(fp,"\n-------------------------------------");
        fclose(fp);
   }          /*   End of file    */
```

Output:

```
        Consumer  Name   : Vivekanandar
       Consumer  No.    : 1000
       Previous reading : 650
       Current  reading : 950
```

After reading the data, calculation will be made and the formatted result will be stored in the specified file **eb.dat** . We can check it by opening the file and it will have the result as below.

```
                    ELECTRICITY  BLL
Name  : Vivekanandar          Service No. : 1000
-------------------------------------------------------
Previous    Current    Units          Amount
Reading     Reading    Consumed       Rs.
-------------------------------------------------------
650.00        950.00       300.00          375.00
-------------------------------------------------------
```

From this example we are familiar in the area of writing formatted result in a file. It is only for one consumer and if we want to produce many no. of bills we can use looping statements. The remaining thing is that how to read the data from the file. Mostly **fscanf( )** function is used to read information from the data file and it may be used for further calculation.

For example the student file **stu.dat** have the information like below. They are read from the file and in order to calculate the class of each student. Type the data and check them in the following program. The data file may be created by any editor.

So the data file **stu.dat** have the informations as

```
Sathya  1000   75
Karthi   1001    35
murugappan  1002  56
mahesh  1003 99
```

```
/* read data and calculate the class, display them */
/* Example for fscanf ( ) function */
#include <stdio.h>
main( )
{  FILE *fp;
  int rollno, marks;
  char name[50],rank[15];
  fp = fopen("stu.dat","r");
  if (fp= =NULL)
    {  printf("\nSorry.  No such file ");
      exit(0);
     }
   printf("\nName       Rollno    Marks    Class\n");
   while (!feof(fp))
    {       fscanf(fp, "%s %d %d", name, &rollno, &marks);
             if (marks>=60)                strcpy(rank, "First");
     if ((marks>=50) && (marks < 60))                strcpy(rank,"Second");
     if ((marks>=40) && (marks <50 ))                strcpy(rank, "Third");
     if (marks<40)               strcpy(rank, "Fail");
    printf("\n%s\t  %6d  %5d  %s",name,rollno,marks,rank);
   }
   fclose(fp);
 }
```

When the above program is executed the data has been read from the file **stu.dat** and displayed as follows.

```
Name       Rollno    Marks    Class
Sathya      1000       75    First
Karthi      1001       35    Fail
murugappan   1002        56    Second
mahesh      1003       99    First
```

Read / write record in the file

The data can be read or write very easily by using some library functions as above. In the previous topics the data are written as characters, lines and formatted input and output statements were used. In this chapter we are going to discuss like how to read or write the record into the file. This way is very useful to write entire structure into the file instead of individual field.

The two functions which are used t perform these operations are

1. fread ( ) - To read a structure
2. fwrite( ) - To write a structure

The function **fread ( )** is used to read a structure from the specified file and **fwrite( )** is used to write a structure into the file. Their syntax are

fread(pointer- to structure**, sizeof** structure**,** no. of records**,** file-pointer)

fwrite(pointer-to structure**, sizeof** structure**,** no. of records**,** file-pointer)

Both of the functions are taking four arguments. They are

1. Pointer to the structure variable
2. Size of the structure
3. No. of structures to read / write
4. File pointer to specify the process on which file

Assume the structure has the following information

```
struct
   {   int  rollno;
        char name[20];
    } stu;
fwrite( &stu, sizeof(stu),1,fp);
```

Here **&stu** is a pointer to the structure, **sizeof(stu)** specifies the memory size of record / structure, No. of structure is **1** and file is **fp.** The following is a simple example for reading and writing structure value in the file.

```
/* Example for structure reading and writing in the file */
#include <stdio.h>
#include <conio.h>
#include <process.h>
#include <string.h>
struct
{   int rollno;
  char name[15];
  float avg;
}stu;
main( )
{ FILE *fp;
   int n,i;
  char res[10];
  clrscr();
  fp=fopen("test.dat","w");
  if (fp= =NULL)
  {   printf("\nCan't create file");
      exit(0);       }
   printf("\nNo. of students :");
   scanf("%d",&n);
   printf("\nEnter the details\n");
 for(i=1;i<=n;i++)
   {       printf("\nRoll No.     :");
           scanf("%d",&stu.rollno);
           printf("\nName         :");
           scanf("%s",stu.name);
           printf("\nAverage      :");
           scanf("%f",&stu.avg);
           fwrite(&stu,sizeof(stu),1,fp);
   }
```

```
        fclose(fp);
        clrscr();
        fp=fopen("test.dat","r");
        printf("\n----------------------------------------");
        printf("\nRollNo.   Name         Average    Result");
        printf("\n----------------------------------------");
        fread(&stu,sizeof(stu),1,fp);
        while(!feof(fp))
          {       if (stu.avg>=40)
                     strcpy(res,"Pass");
                  else
                     strcpy(res,"Fail");
                  printf("\n%4d",stu.rollno);
                  printf("\t%-15s",stu.name);
                  printf("\t%5.2f ",stu.avg);
                  printf("\t   %s",res);
                  fread(&stu,sizeof(stu),1,fp);
      }
      printf("\n----------------------------------------");
          fclose(fp);     getch( );   }
```

Output:

```
No. of students :2
Enter the details
Roll No.     :1001
Name         :karthi
Average      :35.50
Roll No.     :1002
Name         :sabari
Average      :50.75
-------------------------------------------------
RollNo.   Name         Average    Result
-------------------------------------------------
1001    karthi         35.50      Fail
1002    sabari         50.75      Pass
-------------------------------------------------
```

Error Handling

As such C programming does not provide direct support for error handling but being a system programming language, it provides you access at lower level in the form of return values. Most of the C or even Unix function calls return -1 or NULL in case of any error and sets an error code **errno** is set which is global variable and indicates an error occurred during any function call. You can find various error codes defined in <error.h> header file.

So a C programmer can check the returned values and can take appropriate action depending on the return value. As a good practice, developer should set errno to 0 at the time of initialization of the program. A value of 0 indicates that there is no error in the program.

The errno, perror() and strerror()

The C programming language provides **perror()** and **strerror()** functions which can be used to display the text message associated with **errno**.

- The **perror()** function displays the string you pass to it, followed by a colon, a space, and then the textual representation of the current errno value.

- The **strerror()** function, which returns a pointer to the textual representation of the current errno value.

Let's try to simulate an error condition and try to open a file which does not exist. Here I'm using both the functions to show the usage, but you can use one or more ways of printing your errors. Second important point to note is that you should use **stderr** file stream to output all the errors.

```
#include <stdio.h>
#include <errno.h>
#include <string.h>
extern int errno ;
int main ()
{
  FILE * pf;
  int errnum;
  pf = fopen ("unexist.txt", "rb");
  if (pf == NULL)
  {
    errnum = errno;
    fprintf(stderr, "Value of errno: %d\n", errno);
    perror("Error printed by perror");
    fprintf(stderr, "Error opening file: %s\n", strerror( errnum ));
  }
  else
  {
    fclose (pf);
  }
  return 0;
}
```

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

```
Value of errno: 2
Error printed by perror: No such file or directory
Error opening file: No such file or directory
```

Random file operation:

The file is used to read or write the data in permanent for further reference. The data may be processed sequentially or randomly. In sequence the operation is from first to last, ie from the beginning of file to the end of file. Suppose we want to process particular record of the file, sequential is not good. Because it will takes lot of time for processing. Alternative way to process the data is by using the random methods. The following functions are used to process the data in random.

1. fseek ( )
2. ftell ( )
3. fgetpos ( )
4. rewind ( )

fseek - To set the file pointer

The function **fseek( )** is used to set the file pointer in the specified location. The format of the fseek is as follows.

```
result      fseek ( file-pointer , location, from where )
```

Where **result** tells whether the operation is success or not.
filepointer is a pointer, which points to the opened file
location no. of location to be adjusted
fromwhere specifies from where the process starts
The **fromwhere** may be any one of these value.
SEEK_SET => Seeks from beginning of the file
SEEK_CUR => Seeks from current location
SEEK_END => Seeks from end of the file

While opening the file for first process, the file pointer always points to the first location of the file. The file pointer can be set to any new location by using the above type of values.
For example the statements

1. fseek (fp, 10,SEEK_SET);
   => From the first location of file, the pointer is adjusted to 11th location. Count from 0th location.
2. fseek (fp, -5, SEEK_END);
   => From the last location of file, the pointer is adjusted to 5th location from end of file.

For example the file "**test.dat**" contains the following data and its locations are illustrated as

Location	0	1	2	3	4	5	6	8	9	10	11	12	13
Value	I		L	O	V	E		I	N	D	I	A	**EOF**

Total no. of characters in that file is **13** including the end of file (**EOF**). EOF is a single character.

After the statement, **fp = fopen("test.dat","r")**, the file pointer **fp** is pointed to the first location and is

Location	0	1	2	3	4	5	6	8	9	10	11	12	13
Value	I		L	O	V	E		I	N	D	I	A	**EOF**

fp

After the statement **fseek(fp, 5 , SEEK_SET)**, the **fp** is adjusted to the 6th (Because 0 to 5) location from beginning of the file and is

Location	0	1	2	3	4	5	6	8	9	10	11	12	13
Value	I		L	O	V	E		I	N	D	I	A	**EOF**

fp

After the statement **fseek(fp, -4 , SEEK_END)**, the **fp** is adjusted to the 9th location from end of file (last character is **EOF**, so 4th location from last is **N**.

Location	0	1	2	3	4	5	6	8	9	10	11	12	13
Value	I		L	O	V	E		I	N	D	I	A	**EOF**

fp

The following is a program to be used to display the remaining characters from the user-specified location. For example, if the user gives 5 as input and all the remaining characters from 5th location will be displayed.

```
/*Display the characters from the user defined location */
```

```
#include <stdio.h>
main( )
 {      FILE *fp=fopen("c:\\temp\\ek.dat","r");
        char ch;
        int n;
        printf("\nNo. of characters from first : ");
        scanf("%d",&n);
        printf("\nRemainig of file  : \n");
                fseek(fp, n , SEEK_SET);     /* Adjusted to the nth location */
         while(!feof(fp))
        {    ch=fgetc(fp);
             printf("%c",ch);
         }}
```

Output is

```
No. of characters from first : 3
Remainig of file  :
OVE INDIA
```

ftell ( ) - to find the location of the file

The function **ftell( )** is used to find the location of the file pointer. By using the **fseek( )** function , file pointer can be adjusted. The format of **ftell( )** function is as follows.

```
long     ftell( filepointer ) ;
```

For example, the file is adjusted to the 5th location by using the function **fseek(fp, 5 , SEEK_SET)** and it is ensured by checking the location of file pointer with the help of **ftell( )**.

A program which is going to display the content of file and its location.

```
/*Content of the file is displayed with its location */
#include <stdio.h>
main( )
{
  FILE *fp=fopen("ek.dat","r");
  char ch;
  long l;
  while(!feof(fp))
   {              printf("\n%c stored at location  %d
                                ",fgetc(fp),ftell(fp));    } }
```

output :

```
I    stored at location    0
     stored at location    1
L     stored at location    2
O     stored at location    3
V     stored at location    4
E     stored at location    5
     stored at location    6
I    stored at location    7
N     stored at location    8
D     stored at location    9
I    stored at location   10
```

A stored at location 11
EOF stored at location 13

Actually the end of file is not marked as the character like **EOF**, it is only for user identification. The **EOF** file is marked by the character **-1**. So, whenever the EOF is occurred the process is terminated from the while loop.

Size of the file – A simple way

How to find the length of the file? One way is to travel up to the end of file. Another way to obtain the length of the file is using the features **ftell( )** function.

```
/* Finding the size of the file */
#include <stdio.h>
main( )
 {    FILE *fp=fopen("ek.dat","r");
     char ch;
     long l;
     fseek(fp,0,SEEK_END);
     l=ftell(fp);
     printf("\nLength is = %d characters",l);
}
```

output:

```
Length is = 13 characters
```

Reversing the content of a file

Are you able to find the reverse of file? Yes. We can reverse. The file can be reversed if we know the length and file pointer adjustment idea. This can be achieved by using the functions **fseek( )** and **ftell( ).**

If calculated every thing, the file pointer is adjusted from end using the **fseek(fp,-i,SEEK_END)** function. Here **fp** is a file pointer, **i** is the location variable and **SEEK_END** is the place to start the process.

```
/* Reversing the content of file */
#include <stdio.h>
main( )
{   FILE *fp;
   int n,i=0;
   char ch;
   fp = fopen("ek.dat","r");
   fseek(fp,0,SEEK_END);  /*Adjusted to the end */
   n=ftell(fp);        /* Getting the location */
   printf("\n File  In Reverse\n");
   while(i<=n)
   {
     fseek(fp,-i,SEEK_END); /* Adjusted to each character */
      i++;
  ch=fgetc(fp);  /* Reading the  character */
  printf("%c",ch);
}}
```

output:

```
File In Reverse
```

rewind( ) – Adjusted to the beginning

The file pointer can be adjusted and located to anywhere in the program. The additional feature of file in C is to relocate the file pointer to the beginning of the file. It is achieved by using the function **rewind( )**.

Suppose we have thousands of records and are going to be processed very often. Normally the data can be processed as in the previous programs. We are in need of searching the items and the item is in different place. How to search them? Start searching from the beginning of file and continue until the end of file. Once again for new set of data we have to search the entire file. What we have to do is the file pointer should be adjusted to the beginning of file by using **fseek(fp,0,SEEK_SET)**. This task can be achieved by simple function called **rewind( )**. Wherever this function is occurred, the specified file pointer is going to be readjusted to the beginning of file.

General format of rewind function is

```
void    rewind ( file-pointer ) ;
```

The argument to this function is which file pointer should be adjusted.

```
/* Example for Rewind function */
#include <stdio.h>
main( )
{   FILE *fp;
  int count,choice;
  char ch1,ch2;
  fp = fopen("ek.dat","r");
 do
 {  printf("\nDo you want to check (0 - Exit)");
   scanf("%d",&choice);
   fflush(stdin);              /* To remove the characters from buffer if any */
   if (choice!=0)
   {  printf("\nCharacter to check :");
    scanf("%c",&ch1);
    count=0;
   while(!feof(fp))
   {      ch2 = fgetc(fp);
    if (ch1==ch2)
       count++;     }
  printf("\n%c has  occured  %d times  ",ch1,count);
  rewind(fp);
 } }while(choice!=0);
 getch();}
```

Out put:

```
KARPAGAM COLLEGE
Do you want to check (0 - Exit)1
Character to check :A
A has occured  3 times
Do you want to check (0 - Exit)1
Character to check :L
L has  occured  2 times
Do you want to check (0 - Exit) 0
```

Command Line Arguments

Operating system is a collection of programs, which is used to control the operations on the computer system. These pre-written programs are called as **commands**. How do we execute these programs or commands? File types like .COM, .BAT and .EXE can be executed from the command prompt. It is a place where the commands can be given to the operating system or program can be executed and it as like below depending upon the operating system.

In DOS

```
C:\ _                    // For C drive
                          Cursor
                                Prompt
```

Other drive or operating systems prompt appears as like

```
A:\  _      //  For  A  drive  - Floppy
H:\  _      // May be networks
  $ _       // In Unix
```

The commands or programs are executed from the command line. For example the commands can be invoked as follows.

C: \ dir

Here the command **dir** is invoked from the command line (C:\). This command returns all the files in the current directory. If we have any executable file (.EXE), it can also be executed from the command line. For example, we have an executable file **test.exe**, and it will be executed from the command prompt as below.

C:\test // test must be an executable file

What are command line arguments?

Argument is a variable, which is used to carry the information to the function. The value can be passed from program and it can be received by the function as below.

```
/* Example for calling and passing arguments */
 main( )
{          display(100); }
void display( int  x)
{     printf("\nX = %d ",x);}
```

In this program the function **display( )** is invoked from the **main( )** with argument **100**, as **display(100)** and it will be received by the parameters of the function.

"*Passing the arguments from the command line is called as command line arguments*" or *passing arguments from command prompt to that program.*

For example, take the **TYPE** command, which is used to display the content of the specified file. For **TYPE** command we have to pass file name as argument as follows

C:\ TYPE test.c

Here **test.c** is a name of the file to be displayed and it is called as command line argument. How to pass arguments to our program? and who will receive these arguments and where it comes?

Who will receive the command line arguments?

All the C program must be with the function **main( )** and the execution also starts at the **main( )** function. The **main( )** is also a function and it is also able to receive the arguments and its prototype declaration is as follows.

main(int count, char *s[15])

Here the **count** contains the no. of arguments and character array variable **s** contains the values of arguments. The arguments are copied in to the string variable in sequence. For example the arguments from the command line is as follows

C:\ test abc def

Here **test** is the name of the file to be executed, **abc** is the first argument and **def** is the second argument to the function **main( )** function. So these are assigned in sequence and separated by the delimiters. How to process these arguments? No. of argument is in the variable **count**, and the list of arguments are in the variable **s**, which is used to keep array of characters. The arguments **test**, **abc** and **def** are stored in sequence like **s[0]**, **s[1]** and **s[2]**. This discussion is illustrated in the following program.

```
/* Example for Command Line Arguments  */
main(int count,char *s[10])
{  int i;
  printf("\nNo. of arguments : %d ",count);
  printf("\nList of arguments");
  for(i=0;i<count;i++)
    printf("\nArgument %d  is %s ",i , s[i]);
}
```

Type the above program and save it as **"cmd.c"**. After converting into **EXE** file (By Pressing F9), the file is executed from the command prompt and its result is as follows.

```
C:\ekarthi\c>cmd  abc  def
        No. of arguments : 3
        List of arguments
        Argument 0  is C:\EKARTHI\C\CMD.EXE
        Argument 1  is abc
        Argument 2  is def
```

Preprocessor in C

The C preprocessor is a *macro processor* that is used automatically by the C compiler to transform your program before actual compilation. It is called a macro processor because it allows you to define *macros*, which are brief abbreviations for longer constructs.

A macro is used to define the short cuts for the programmer. For example we have to use the string "Karpagam Arts and Science College" in many places. It takes time for type. Instead of repeating the same we can define some shortcut name for this string and it can be used instead of entire string.

One more advantage when we use shortcuts. Suppose we need to change content of the string or few characters in that. In this case instead of changing in every places, changes in macro definition will affect whole. It is also used to include the content of another file to the currently used file. Like this there are so many features in the macro. We will discuss about them one by one. The macro definition is the first statement to be executed and it is started by the character #.

#define - for simple definition

It is used to define the shortcuts, simple formulas and functions also. The format is

```
#deinfe   shortcut   actual value
```

In this case, instead of actual value, shortcut can be used in every where and at the time of compilation its actual character replaces shortcuts. There is no semicolon at the end of the macro definition statement. For example

```
#define name "Karpagam Arts and Science College"
```

Now instead of the string "Karpagam Arts and Science College", the shortcut characters college can be used. The changes in the string also will affect in all the places.

```
#define name "Karpagam Arts and Science College"
main ()
{       printf(name);  }
```

Some simple example for #define statements are given below.

```
#define   cls      clrscr()
#define   nl       printf("\n")
#define   getInt(n)  scanf("%d", &n)
```

In the last declaration, instead of complex **scanf( )** statement **getInt( )** can be used.

Another example for definition is that to define a simple function. To reduce the size of the program we can utilize this feature. But it takes more long steps to implement in the program. By using the macro, the small function can be written easily as below.

```
/* Example for Macro */
#define big(a,b)  a>b?a:b
void main()
{ int a=210,b=20;
        printf("\nBig : %d ",big(a,b));
        getch();          }
```

When we use the **big(a,b)**, it will be replaced by the statement **a>b?a:b** So, it can be used for defining simple functions.

A macro allows declaring more than one statement as below. Here the line(n) definition contains group of statements.

```
int n,i;
#define line(n) for(i=1;i<n;i++)  printf("*");
#define string "\nKarpagam  College"
void main()
{
        line(35);
        printf(string);
        line(35);
        getch();
 }
```

CODE TO SHUT DOWN THE COMPUTER-WINDOWS

```
#include <stdio.h>
#include <stdlib.h>
 main()
{   char ch;
   printf("Do you want to shutdown your computer now (y/n)\n");
  scanf("%c",&ch);
   if (ch == 'y' || ch == 'Y')
     system("C:\\WINDOWS\\System32\\shutdown -s");
   return 0;
}
```

Class: I B.Sc IT **Course Name: Programming Fundamentals Using C/C++**
Course Code: 18ITU101 **Unit: IV (Memory Allocation, File I/O)** **Batch 2018-2021**

Possible Questions

Part-A (1 mark-Online Exam)

Part-B (2 marks)

1. What is a file?
2. Describe about the categories of preprocessor directives.
3. What is a preprocessor?
4. Explain the following i) #define ii) #include
5. What is the use of file inclusion?
6. How will you read, write and append data in a file
7. What is the significance of EOF?
8. What are Random access files?

Part-C (6 marks)

1. Write in detail about different formats for opening a file. Explain with example.
2. Explain about command line arguments with a sample program.
3. Write in detail about conditional compilation using preprocessors
4. Write a program to copy the content of a file into another using command line argument?
5. Write in detail about macro substitution with example.
6. Explain the following file functions a) ftell b)fseek c)rewind
7. Write a program to create a file for employee pay slip and manipulate it?

KAHE

KARPAGAM ACADEMY OF HIGHER EDUCATION					
PART - A (ONLINE EXAMINATION)					
MULTIPLE CHOICE QUESTIONS (Each question carries one mark)					
SUBJECT: PROGRAMMING FUNDAMENTALS USING C/C++					
QUESTIONS	OPTION1	OPTION2	OPTION3	OPTION4	**ANSWER**
In ____ mode the existing file is opened for reading only	r	w	a	f	r
In _____ mode the file can be opened for writing only	r	w	a	f	w
In _____ mode the file can be opened for appending data to it	r	w	a	f	a
The getc function will return an _____ ,when end of the file has been reached	BOF	EOF	SOF	FOF	EOF
In fseek function value of offset should be ____ to move the pointer to beginning of file	0	1	2	3	0
In fseek function value of offset should be ____ to move the pointer to current position	0	1	2	3	1
In fseek function value of offset should be ____ to move the pointer to end of file	0	1	2	3	2
_____ is a parameter supplied to a program when the program is invoked	argument	parameter	command line argument	values	command line argument
Command line argument is supplied to the program when it is _____	invoked	developed	compiled	stored	invoked
The file mode _____ is used to read and append some data into an existing file from end of the file	"r"	"r-"	"r+"	all	"r+"
A _____ file is a collection of ASCII characters, with end of line markers and end of file markers	program	binary	image	text	text
The files opened with mode “w allows	reading and writing	reading only	writing only	reading and appending	writing only
The files opened with mode “a+” allows	reading and writing	reading only	writing only	reading and appending	reading and appending
The _____ function can be used to test for an end of file condition	eof()	feof()	eol()	EOF	feof()
The negative integer constant EOF indicates the ____	end of line	new line	end of file	minimum integer range	end of file
fclose() function is used to close	editor	program	all the above	file	file

File mode must be specified while _____	opening a file	reading a file	writing a file	closing a file	opening a file
_____ function is used to write a string into an ASCII file	fwrite()	writes()	puts()	fputs()	fputs()
_____ is used to read a number of items from the file stream using format	printf()	scanf()	fprintf()	fscanf()	fscanf()
____ function is used to report the status of the file and returns a non zero integer if an error has been detected	fstatus()	ferror()	fnull()	ifzero()	ferror()
fseek, ftell and rewind functions are used with _____ files	sequential files	indexed files	random access files	all files	random access files
_____ returns the current position of the file pointer in a file	pos()	fseek()	ftell()	fposition()	ftell()
_____ is used to move the file pointer to the desired location in a file	pos()	fseek()	ftell()	fposition()	fseek()
_____ takes the filepointer and reset the position to the start of the file	pos()	fseek()	ftell()	rewind()	rewind()
Which of the following functions will take only the file pointer as its argument	fclose()	ferror()	rewind()	all of the above	all of the above
_____ function is used to read some bytes from the binary files	read()	readln()	readf()	fread()	fread()
_____ function is used to write some bytes into a binary files	writebytes()	fwrite()	writef()	putfile()	fwrite()
main() function takes _____ number of arguments	one	two	three	any	two
The _____ in the main(argc, argv) function represents an array of character pointers that points to the command line arguments	argc	argv	args	cptr	argv
In main(argc,argv),the variable argc ____	counts the number of arguments in	b) counts the number of functions in a	arranges the argument in the command	sets a pointer to the argument in	counts the number of arguments in command line
In the command line argv[0] points to the _____	program under execution	first argument after the program name	the beginning of the program file	all the elements in the arguments	program under execution
Command line arguments are used to accept argument from	command prompt of operating	through scanf() statement	through printf() statement	through gets function	command prompt of operating system
The malloc() function	allocates memory and not returns a	allocates memory and returns a	changes the size of allocated	deallocates or frees the memory	allocates memory and returns a pointer to the first byte of it
The _____ processes the source code before it passes through the compiler	interpreter	preprocessor	linker	assembler	preprocessor
Preoprocessor directives must be present	before the main () function	after the main() function	at the end of the program	anywhere in the program body	before the main () function
When files are included using #include<filename> ,then the file is searched in_____	standard library only	current directory only	in all directories	current directories & in standard	standard library only
FILE *fp; fp=fopen(“ fn”,”m”); In the above syntax fp is a _____	file name	file variable	pointer to data type FILE	pointer to function fopen()	pointer to data type FILE
_____ is analogous to getchar() function and reads a character from a file	getch()	gets()	getc()	getw()	getc()
_____is functions used to write integers into a file	putw()	puts	puti()	putc	putw()

KARPAGAM ACADEMY OF HIGHER EDUCATION
(Deemed to be University)
(Established Under Section 3 of UGC Act 1956)
COIMBATORE – 641 021

INFORMATION TECHNOLOGY/COMPUTER TECHNOLOGY

First Semester
FIRST INTERNAL EXAMINATION - July 2018

PROGRAMMING FUNDAMENTALS USING C/C++

Class & Section: I B.Sc (IT) & I B.Sc CT **Duration: 2 hours**
Date & Session : **Maximum marks: 50 marks**
Subj.Code: 18ITU101

PART- A (20 * 1= 20 Marks)
Answer ALL the Questions

1. _____ refers to finding value that do not change during execution of program.
   a. keyword b. identifier c. constant d. token
2. _____ constant contains single character enclosed within pair of single quote marks.
   a. string b. variables c. character d. numeric
3. _____ is data name that may be used to store a data value.
   a. string constant c. character
   b. variables d. numeric
4. Characters are usually stored in _____ bits.
   a. 8 b. 16 c. 24 d. 32
5. Floating point numbers are stored in _____ bits.
   a. 8 b. 16 c. 24 d. 32
6. _____ operator is used for manipulating data at bit level.
   a. logical a. bitwise b. arithemetic c. sizeof
7. Execution of C Program begins from _____ function.
   b. user defined d. header file
   c. main e. statements
8. Every C program must have exactly _____ main function
   a. one b. two c. three d. none
9. The variables are initilaised using _____ operator
   a. > b. = c. ?= d. +
10. In ASCII character set the uppercase alphabet represent codes _____
    a. 65 to 90 b. 96 to 45 c. 97 to 123 d. 1 to 26
11. Individual values in array is referred as _____
    a. subscript b. elements c. subelements d. pointers
12. Any subscript between _____ are valid for an array of fifty elements
    a. 0-49 b. 0-50 c. 0-47 d. 0-51
13. Value in a matrix can be represented by _____ subscript
    a. 1 b. 3 c. 2 d. 4
14. Arrays that do not have their dimensions explicitly specified are called_____
    a. unsized arrays c. initialized arrays
    b. undimensional arrays d. to size arrays
15. printf belongs to the category ____ function
    a. user defined c. subroutine
    b. library d. preprocessor

16. Conditional operator is a combination of ___ and _____ operator
    a. ?: b. &? c. &* d. ?,
17. Case Labels end with ____ operator
    a. : b. ; c. . d. ,
18. In switch statement if the value of the expression does not matches with any of the case values _____ is executed
    a. optional b. case c. default d. loop
19. While loop is _____ statement
    a. entry controlled
    b. exit controlled
    c. branching
    d. none of the above
20. Do.. While loop is _____ statement
    a. entry controlled
    b. exit controlled
    c. branching
    d. none of the above

PART- B (3 * 2= 6 Marks)
Answer ALL the Questions

21. List the primary data types in C and give examples for each.
22. Define Variable. List the rules to be followed for framing variable name.
23. If a character string is to be received through the keyboard which function would work faster? Why?

PART C (3 * 8 = 24 Marks)
Answer ALL the Questions

24. a. Explain in detail about the operators used in C. Provide examples for each.

(OR)

    b. Explain the data types used in C with example
25. a. Explain conditional control statements with neat example.

(OR)

    b. List the looping statements. Explain each with neat example.
26. a. How will you declare and initialize a two dimensional array? Write a C program to perform matrix addition.

(OR)

    b. What is an array? Explain in detail the various types of arrays with example.

KARPAGAM ACADEMY OF HIGHER EDUCATION					
PART - A (ONLINE EXAMINATION)					
MULTIPLE CHOICE QUESTIONS (Each question carries one mark)					
SUBJECT: PROGRAMMING FUNDAMENTALS USING C/C++					
QUESTIONS	OPTION1	OPTION2	OPTION3	OPTION4	**ANSWER**
The wrapping up of data & function into a single unit is known as _____	Polymorphis	encapsulation	functions	data members	encapsulation
_____refers to the act of representing essential	encapsulatio	inheritance	Dynamic binding	Abstraction	Abstraction
Attributes are sometimes called_____	data members	methods	messages	functions	data members
The functions operate on the datas are called_____	methods	data members	messages	classes	methods
_____is the process by which objects of one class acquire the	polymorphis	encapsulation	data binding	Inheritance	Inheritance
_____means the ability to take more than one form	polymorphis	encapsulation	data binding	none	polymorphism
The process of making an operator to exhibit different behaviors	function overloading	operator overloading	method overloading	message overloading	operator overloading
Single function name can be used to handle different types of tasks is known as	function overloading	operator overloading	polymorphi	encapsulation	operator overloading
_____means that the code associated with a given	late binding	Dynamic binding	Static binding	none	Dynamic binding
Objects can be_____	created	created & destroyed	permanent	temporary	created & destroyed
_____helps the programmer to build secure programs	Dynamic binding	Data hiding	Data building	message passing	Data hiding
_____techniques for communication between	message passing	Data binding	Encapsulati	Data passing	message passing
_____refers to the use of same thing for different purposes	overloading	Dynamic binding	message loading	none	overloading
_____are extensively used for handling class objects	overloaded functions	methods	objects	messages	overloaded functions
_____is used to reduce the number of functions to be defined	default arguments	methods	objects	classes	default arguments
_____ enables an object to initialize itself when it is created	Destructor	constructor	overloading	none of the above	constructor
_____ destroys the objects when they are no longer required	Destructor	constructor	overloading	none of the above	Destructor
The _____ is special because its name is the same as the class name.	Destructor	static	constructor	none of the above	constructor
Constructors are invoked automatically when the _____ are created	Datas	classes	objects	none of the above	objects
Constructors cannot be _____	Inherited	destroyed	both a & b	none of the above	Inherited
When more than one constructor function is defined in a class,	Multiple	copy	default	overloaded	overloaded
C++ complier has a _____ constructor, which creates objects, even though it was not	Explicit	default	implicit	none of the above	implicit

A _____ constructor is used to declare and initialize an object from another object	Default	copy	multiple	parameterized	copy
A destructor is preceded by _____ symbol	Dot	asterisk	colon	tilde	tilde
The class can have only _____ destructor	two	many	one	four	one
In overloading of binary operators, the _____ operand is used to invoke the operator function.	Right-hand	Arithmetic	Left-hand	Multiplication	Left-hand
_____ functions may be used in place of member functions for overloading a binary	Inline	Member	Conversion	Friend	Friend
The operator that cannot be overloaded is _____	Single of	+	-	=	Single of
The overloading operator must have atleast _____ operand that is of user-defined data	Two	Three	One	Four	One
Operator overloading is called _____	Function Overloading	Compile time polymorphism	Casting operator	Temporary object	Compile time polymorphism
The mechanism of deriving a new class from an old one is called _____	Operator overloading	Inheritance	Polymorphi	Access mechanism	Inheritance
_____ provides the concept of reusability.	Overloading	Message passing	Data abstraction	Inheritance	Inheritance
A derived class with only one base class is called _____ inheritance.	Single	Multi-level	Multiple	Hierarchical	Single
The derived class inherits some or all of the properties of _____ class.	Member	Base	Father	Ancestor	Base
The _____ class inherits some or all of the properties of base class.	Abstract class	Father class	Derived class	Child class	Derived class
A class that inherits properties from more than one class is known as _____ inheritance.	Multiple	Multilevel	Single	Hybrid	Multiple
The class that can be derived from another derived class is known as _____ inheritance.	Hierarchical	Single	Multi-level	Hybrid	Multi-level
When the properties of one class are inherited by more than one class, it is called _____	Single	Hybrid	Multiple	Hierarchical	Hierarchical
A private member of a class cannot be inherited either in public mode or in _____ mode.	Private	Protected	Visibility	Nesting	Private
A protected member inherited in public mode becomes _____	Highly protected	Private	Public	Protected	Protected
A protected member inherited in private mode becomes _____	Visibility	Private	Protected	Public	Private
The _____ are called as overloaded operators	>> and <<	+ and –	* and &&	– and .	>> and <<
The >> operator is overloaded in the _____	istream	ostream	iostream	None	istream
Templates are suitable for _____ data type.	any	basic	derived	all the above	basic
Templates can be declared using the keyword _____	class	template	try	none	template
Templates is also called as _____ class.	generic	container	virtual	base	generic

Function Templates can accept only _____ parameters.	one	any	two	none.	any
Select the correct Template definition _____ .	template <class T>	class <template T>	template <T>	template class <T>.	template <class T>
Function Templates are normally defined _____ .	in main function	globally	in a class	anywhere	in a class
The statement catches the exception _____ .	catch	try	template	throw.	catch
In a multiple catch statement the number of throw statements are .	same as catch	twice than catch	only one	none.	only one
The exception is generated in _____block.	try	catch	finally	throw.	try
The exception handling one of the function is implicitly invoked.	abort	exit	assert	none.	abort
The exception handling mechanism is basically built upon _____ keyword	try	catch	throw	all the above	all the above
The point at which the throw is executed is called _____.	try	catch	throw point	none	throw point
A template function may be overloaded by _____ function	template	ordinary	both (a)and (b).	none	both (a)and (b).

KARPAGAM ACADEMY OF HIGHER EDUCATION
(Deemed to be University)
(Established Under Section 3 of UGC Act 1956)
COIMBATORE – 641 021

INFORMATION TECHNOLOGY/COMPUTER TECHNOLOGY

First Semester
FIRST INTERNAL EXAMINATION - July 2018

PROGRAMMING FUNDAMENTALS USING C/C++

Class & Section: I B.Sc (IT) & I B.Sc CT **Duration: 2 hours**
Date & Session : **Maximum marks: 50 marks**
Subj.Code: 18ITU101

PART- A (20 * 1= 20 Marks)
Answer ALL the Questions

1. _____ refers to finding value that do not change during execution of program.
   a. keyword b. identifier **c. constant** d. token
2. _____ constant contains single character enclosed within pair of single quote marks.
   a. string b. variables **c. character** d. numeric
3. _____ is data name that may be used to store a data value.
   a. string constant c. character
   b. variables d. numeric
4. Characters are usually stored in _____ bits.
   a. 8 b. 16 c. 24 d. 32
5. Floating point numbers are stored in _____ bits.
   a. 8 b. 16 c. 24 **d. 32**
6. _____ operator is used for manipulating data at bit level.
   a. logical **a. bitwise** b. arithemetic c. sizeof
7. Execution of C Program begins from _____ function.
   b. user defined d. header file
   c. main e. statements
8. Every C program must have exactly _____ main function
   a. one b. two c. three d. none
9. The variables are initilaised using _____ operator
   a. > **b. =** c. ?= d. +
10. In ASCII character set the uppercase alphabet represent codes _____
    a. 65 to 90 b. 96 to 45 c. 97 to 123 d. 1 to 26
11. Individual values in array is referred as _____
    a. subscript **b. elements** c. subelements d. pointers
12. Any subscript between _____ are valid for an array of fifty elements
    a. 0-49 b. 0-50 c. 0-47 d. 0-51
13. Value in a matrix can be represented by _____ subscript
    a. 1 b. 3 **c. 2** d. 4
14. Arrays that do not have their dimensions explicitly specified are called_____
    a. unsized arrays c. initialized arrays
    b. undimensional arrays d. to size arrays
15. printf belongs to the category ____ function
    a. user defined c. subroutine
    b. library d. preprocessor

16. Conditional operator is a combination of ___ and _____ operator
    a. ?: b. &? c. &* d. ?,
17. Case Labels end with ____ operator
    a. : b. ; c. . d. ,
18. In switch statement if the value of the expression does not matches with any of the case values _____ is executed
    a. optional b. case **c. default** d. loop
19. While loop is _____ statement
    a. entry controlled c. branching
    b. exit controlled d. none of the above
20. Do.. While loop is _____ statement
    a. entry controlled c. branching
    b. exit controlled d. none of the above

PART- B (3 * 2= 6 Marks)
Answer ALL the Questions

21. List the primary data types in C and give examples for each.
    Char
    Int
    Long int
    Float
    Double
    Long double
    Unsigned char
    Unsign int
    Unsign long int
22. Define Variable. List the rules to be followed for framing variable name.
    A variable is a data name that may be used to store a data value. A variable may take
    different values at different times of execution and may be chosen by the programmer in a
    meaningful way. It may consist of letters, digits and underscore character.
    Eg: 1) Average
    2) Height
    Rules for defining variables
    - They must begin with a letter. Some systems permit underscore as the first character.
    - ANSI standard recognizes a length of 31 characters. However, the length should not be normally more than eight characters.
    - Uppercase and lowercase are significant.
    - The variable name should not be a keyword.
    - White space is not allowed.

23. If a character string is to be received through the keyboard which function would work faster? Why?
    gets() function to be used

PART C (3 * 8 = 24 Marks)
Answer ALL the Questions

24. a. Explain in detail about the operators used in C. Provide examples for each.

Operators and expressions

C programming language provides several operators to perform different kind to operations. There are operators for assignment, arithmetic functions, logical functions and many more. These operators generally work on many types of variables or constants, though some are restricted to work on certain types. Most operators are binary, meaning they take two operands. A few are unary and only take one operand

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

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Assignment Operators
- Misc Operators

Arithmetic Operators

Following table shows all the arithmetic operators supported by C language. Assume variable **A** holds 10 and variable **B** holds 20 then:

Operator	Description	Example
+	Adds two operands	A + B will give 30
-	Subtracts second operand from the first	A - B will give -10
*	Multiplies both operands	A * B will give 200
/	Divides numerator by de-numerator	B / A will give 2
%	Modulus Operator and remainder of after an integer division	B % A will give 0
++	Increments operator increases integer value by one	A++ will give 11
--	Decrements operator decreases integer value by one	A-- will give 9

Relational Operators

Following table shows all the relational operators supported by C language. Assume variable **A** holds 10 and variable **B** holds 20, then:

Operator	Description	Example
==	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A == B) is not true.
!=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

Logical Operators

Following table shows all the logical operators supported by C language. Assume variable **A** holds 1 and variable **B** holds 0, then:

Operator	Description	Example
&&	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.

Bitwise Operators

Bitwise operator works on bits and perform 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

Assume if A = 60; and B = 13; now in binary format they will be as follows:
A = 0011 1100
B = 0000 1101

A&B = 0000 1100
A|B = 0011 1101
A^B = 0011 0001
~A = 1100 0011

The Bitwise operators supported by C language are listed in the following table. Assume variable A holds 60 and variable B holds 13, then:

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12, which is 0000 1100
|	Binary OR Operator copies a bit if it exists in either operand.	(A | B) will give 61, which is 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) will give 49, which is 0011 0001
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) will give -61, which is 1100 0011 in 2's complement form.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 will give 240 which is 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15 which is 0000 1111

Assignment Operators

There are following assignment operators supported by C language:

Operator	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand	C = A + B will assign value of A + B into C
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand	C %= A is equivalent to C = C % A
<<=	Left shift AND assignment operator	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator	C &= 2 is same as C = C & 2
^=	bitwise exclusive OR and assignment operator	C ^= 2 is same as C = C ^ 2
|=	bitwise inclusive OR and assignment operator	C |= 2 is same as C = C | 2

Misc Operators : sizeof & ternary

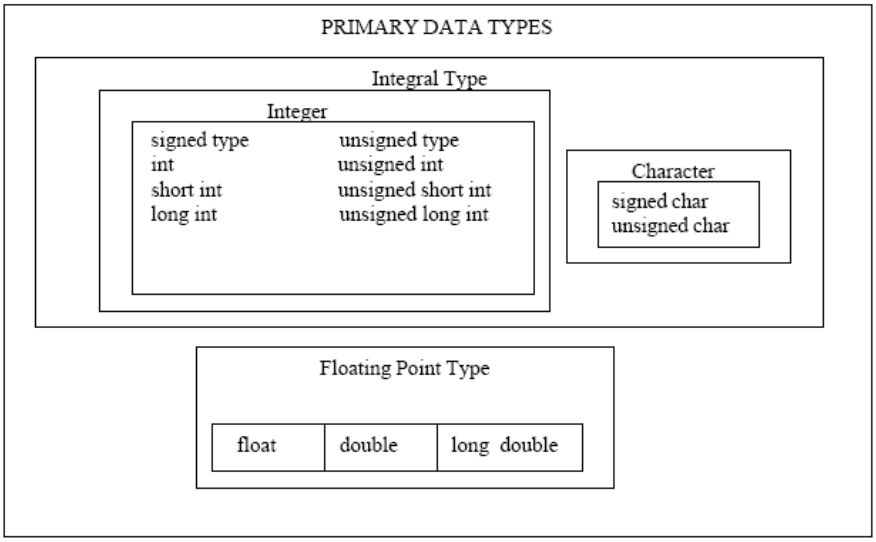
There are few other important operators including **sizeof** and **? :** supported by C Language.

Operator	Description	Example
sizeof()	Returns the size of an variable.	sizeof(a), where a is integer, will return 4.
&	Returns the address of an variable.	&a; will give actual address of the variable.
*	Pointer to a variable.	*a; will pointer to a variable.
? :	Conditional Expression	If Condition is true ? Then value X : Otherwise value Y

(OR)

b. Explain the data types used in C with example

The type of data which a variable can store is called its data type. C language supports following data types:-



Keyword	Description	Low	High	Bytes	Format string
Char	Single character	-128	127	1	%c
Int	Integer	-32768	32767	2	%d
Long int	Long int	-2147483648	2147483848	4	%ld
Float	Floating	3.4 e-38	3.4 e 38	4	%f
Double	Double floating	1.7 e -308	1.7 e 308	8	%lf
Long double	Long double floating	3.4 e -4932	1.1 e 4932	10	%Lf
Unsigned char	Char with no sign	0	255	1	%c
Unsign int	Int with no sign	0	65535	2	%u
Unsign long int	Long int no sign	0	4294967295	4	%lu

25. a. Explain conditional control statements with neat example.

There are few control statements available and they are

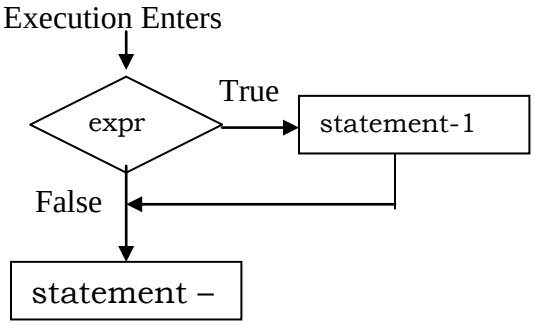
1. Simple if
2. If – else
3. Nested if
4. Switch

1. Simple if - for Small Comparison

This control statement is used to check a condition and based on the result the order of execution will be changed. The generals format and flow-chart for a simple **if** statement is as follows.

```
if  ( expr)          // A Place to check the condition
    {
      statement-1;
    }
statement-n;
```

Flowchart is



Here **expr** is an expression and the result of the **expr**ession may be **TRUE / FALSE.**
If the result of expression is **TRUE,** then the **statement-1** part will be executed. Otherwise the control jumps to the **statement-n** part and continues the execution.The **statement-1** can be a simple statement or a compound statement. The compound statement must be enclosed with in braces **{ }**.

```
/* Example for simple if statement    */
        main( )
        {
           int  a;
           printf("\nEnter a number");
       scanf("%d",&a);
           if (a>0)
                printf("\nThe number is positive");
        }
```

Output:

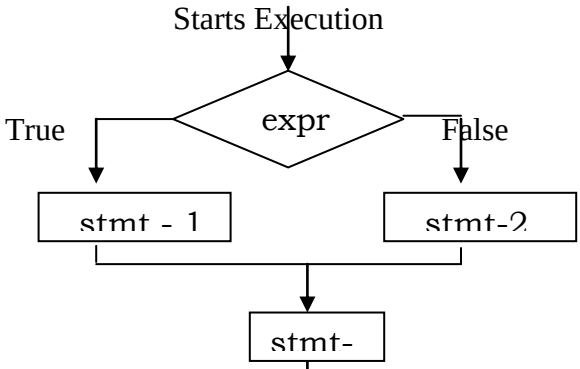
```
Enter a number  5
The number is  positive
Enter a number  -5
```

2 if - else

By using the simple **if** statement we can execute only one set of statement(s) depending upon the condition **TRUE/FALSE**. What we should do? If there are two possible results (TRUE/FALSE). We have a solution with another type of control statement **if-else.** The general format of **if-else** is illustrated with flowchart is as follows.

```
if (expr)
   stmt –1;            // TRUE Part
else
   stmt-2;             // FALSE Part
stmt-n;
```

Flow chart



First the expression **expr** will be executed and if the result is **TRUE**, then the **stmt-1** part will be executed, otherwise **stmt-2** will be executed. There is no chance to execute both statements (**stmt-1** and **stmt-2**) simultaneously. After the execution of any **stmt-1** (or) **stmt-2** process continues from the **stmt-n.** The statements may be simple or compound statements.

```
/* Example for if - else statement is here. */
main( )
{   int n;
   printf("\nEnter a number to check");
   scanf("%d", &n);

   if (n%2 = = 0)
         printf("\n%d is an even number",n);
   else
         printf("\n%d is an odd number",n);
}
```

Output:

```
Enter a number to check 5
5 is an odd number
```

The value of **a** and **b** are compared using the relational operator >. When result of **a>b** is **TRUE,** value of **a** will be assigned to the variable **big,** otherwise the value of **b** will be assigned to **big**. Finally the value of variable **big** will be printed which holds the biggest of two numbers.

Another example to find biggest of three numbers.

---

3. Nested if statement - To check more conditions

Using the previous type of **if** statements we can check the conditions at only one place. Alternatively we can check more conditions using the logical operators. Suppose there is a situation to check number of conditions, in different part of **if** statement, we can use the **nested if** statement. That is, the **if** statement contains another **if** statement as its body of the statement.

Format –1 Format – 2

```
if    (expr1)
      if (expr2)
        statement-1;
      else
          statement-2;
```

```
if    (expr1)
      if (expr2)
         statement-1;
     else
          statement-2;
else
          statement-3;
```

The execution form of **format-1**:

*) First the **expr1** is evaluated in both the format

*) If the result is **TRUE**, then the **expr2** will be evaluated. If **expr2** also returns **TRUE,** the **statement-1** will **be** executed.

*) If the result of **expr-2** is **FALSE** the **statement-2** will be executed.

The execution form of **format-1**:

*) As in **format-1, expr1** is evaluated first

*) if the **expr1** and **expr2** are **TRUE** then **statement-1 w**ill be executed.

*) if the **expr1** is **TRUE** and **expr2** is **FALSE** the **statement-2** will be executed.

*) if the **expr1** is **FALSE** the **statement-3** will be evaluated.

Note: *Every **else** is closest to it's **if** statement.*

Thus of statement can be used inside the body of another if statement (nesting of ifs). The following program is an example for **nested if,** to find the biggest among three numbers.

```
/* Example for nested if statement */
main ( )
{       int a,b,c;
        int big;
        printf("\nEnter three numbers : ");
        scanf("%d%d%d",&a,&b,&c);
        if (a>b)
                if (a>c)
                        big=a;
                else
                        big=c;
        else
                if (b>c)
                        big=b;
                else
                        big=c;

        printf("\nBigget no. is : %d ",big);
        getch( );
}
```

4 Switch – Case Statement:

Whenever the situation occurs like to check more possible conditions for single variable, there will be a no. of statements are necessary. the **switch-case** statement is used to check multiple conditions at a time, which reduces the no. of repetition statements. The general format of switch-case is as follows.

```
switch (expr)
{
     case  c-1:          statement-1;

     case  c-2:          statement-2;

    case  c-4:           statement-4;

    [ default  :   statement-n;  ]
}
```

Way of Execution:

*) First the **expr** will be evaluated and it must return a constant value. The constant can be numeric or character.

*) The result of **expr** is checked against the constant values like **c-1, c-2,** etc., and if any value is matching, the execution starts from that corresponding statement. The execution will continue until the end of **switch** statement.

To avoid this continuous execution problem we can use the **break** statement. The **break** is used to terminate the process of block like switch, looping statements. Here the **default** is an optional statement in switch. If no matching has occurred, the **default** part of statement will be executed and may occur any where in the switch statement.

```
/* Example for switch statement without break */
main( )
{
   int  a;
   printf("\nEnter  any value  for  a : ");
   scanf("%d",&a);
   switch(a)
   {
        case  1 :        printf("\nGood");
        case  2 :        printf("\nWell");
        case  3 :        printf("\nExcellent");
        default :  printf("\nBad  Guy");
   }
}
```

Output:

```
Enter any  value  for  a :  2
Well
Excellent
Bad Guy
Enter any  value  for  a :  5
```

Bad Guy

```
/* Example for the importance of break statement */
main( )
{
   int  a;
   printf("\nEnter  any value  for  a : ");
   scanf("%d",&a);

   switch(a)
   {
        case  1 : printf("\nGood");  break;
        case  2 : printf("\nWell");    break
        case  3 : printf("\nExcellent");  break
        default : printf("\nBad  Guy");
  }
}
```

Output:

```
Enter any  value  for  a :  1
Good
Enter any  value  for  a :  5
Bad Guy
```

Upon using the break statement the statement corresponding to the matching case is only executed. The last statement or default statement is not in need of **break**, because there is no more statements for further execution in the switch.

(OR)

b. List the looping statements. Explain each with neat example.

Looping Statements:

The simple statements that we have discussed so far are used to execute the statements only once. Suppose a programmer needs to execute the specified statements multiple times. Here comes the looping statement, which overcomes the no. of times. For example, to print the string **"Good"** five times, we have to use five individual **printf( )** statements. But, imagine if the no. of time increases to print 1000 times or N times. By using the looping statements a statement or set of statements can be executed repeatedly.

```
main( )
{
        for(i=1;i<=100;i++)
            printf("\nGood morning");
}
```

Just think about the no . of statements in the above programs, with and without looping statement. So, we can use the looping statement to reduce the size of the program.

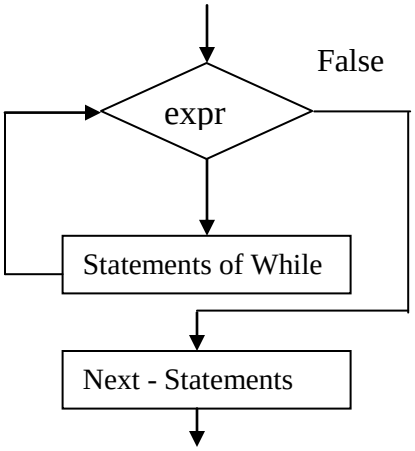
The different types of looping statements are

1. While statement
2. Do-While statement
3. for statement

1 while statement:

While is an **entry controlled** looping statement used to execute it's body of the statement any no. of times. The general format of while statement and its flowchart is as follows .

```
while ( expr)
{
  //Statmts  of  while;
}
  next-statement;
```



Way of execution:

- First the **expr** is evaluated, which will yield **TRUE** or **FALSE** result.
- If the condition is **TRUE,** the control enters inside the **statements of while** and after completion of these statements once again the condition will checked with new value for the next execution.
- Entry of the loop will be determined by the condition, so it is also called entry controlled looping statement.
- So the **Body of while** is executed until the condition becomes **FALSE.**
- If the condition is **FALSE**, execution jumps to the **next-statement** after the **while** statement and continues the execution.

The following example prints the numbers from 1 to n using **while** loop.

```
/* Example  for while loop  */
main( )
{
        int i=1,n;
        printf("\nHow many numbers");

        scanf("%d",&n);
        printf("\nThe numbers are ");
        while(i<=n)
        {
          printf("%5d",i);
          i++;
        }
}
        /* Reverse the given Integer number */
main( )
{
        int n;
        printf("\nEnter a number    :");
        scanf("%d",&n);
        printf("\nReverse of number :");
        while(n != 0)
```

```
        {
                printf("%d",n%10);
                n=n/10;
        }
}
```

2 Do-While statement

It is also a looping statement to execute the specified statements repeatedly (or) any number of times. In the **while** loop, condition is checked first, and execute it's statement only when the result is **TRUE**.

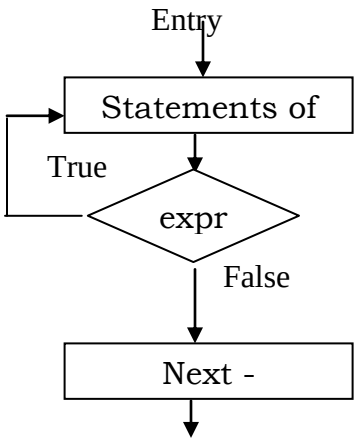
What is the difference between while & do-while?

while is a entry controlled looping statement and the statement of **while** will be executed only when the condition is **TRUE**. But in **do-while**, statement part will be executed first then only the condition will be checked. So the condition may be **TRUE** / **FALSE,** but the statement part will execute at least once.

The general format and it's flowchart is given below.

```
do
{
   Body of   loop;
}
while(expr);

next-statement;
```



Here, the Body of loop will be executed first and the expression **expr** will be checked after the execution of the statement parts. If the result of **expr** is **TRUE** the control starts its execution once again from Body of loop. This process will continuous until the result of **expr** is **FALSE**.

The following program is like a menu selection program.

```
/* Example for do-while looping statement */
main( )
{int ch;
        do
        {
                printf("\n1. Add\n2. Sub");
                printf("\nSelect a choice");
                scanf("%d",&ch);
        }
        while(ch<3);
 }
```

Output :

```
1. Add
2. Sub
Select a choice  1
1.  Add
2.  Sub
Select a choice  3
```

The following is another example for Decimal to Binary conversion.

```
/* Converting Decimal no. To binary  */
main( )
{
        int a,n,s=0,i=1;
        printf("\nDecimal No . :");
        scanf("%d",&n);
        printf("\nBinary  No . :");
        while(n)
        {
           printf("%d",n%2);
           n=n/2;
        }
}
```

Output:

```
Decimal No . :5
Binary  No . :101
```

3 For statement – Flexible looping statement

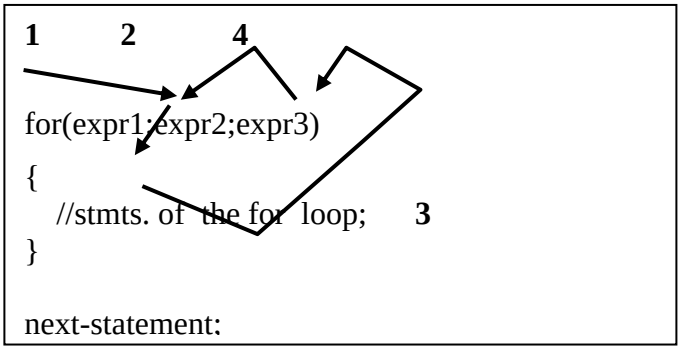
It is also a looping statement to execute the specified statements repeatedly in a simplified format than the previous loops.

In general all the looping statements have the following three steps (Parts) in a loop

1. Initialize the loop control variable
2.Check the condition whether TRUE / FALSE
3. Modify the value of the loop control variable for next execution

In the previous type of looping statements, these statements are kept separately. But in **for** statement all the three parts are kept in one place.

The general format and flowchart is given below.



Where

*) **expr1** is used to initialize the value for loop control variable

*) **expr2** is used to check condition

*) **expr3** is used to modify the value

The **expr1,** step 1 is the statement executed first and only once in the looping statement. The steps 2,3 and 4 will be executed continuously until the condition becomes false, at 2nd place. 2,3,4 is the sequence of execution in the for loop.

Way of execution:

*) First the value of loop control variable is initialized by **expr1.**

*) Next the condition is checked by **expr2** and if it is **TRUE** then the **body of loop** will be executed otherwise the control passes to the **next-statement** of the program.

*) For every **Body of loop** execution, **expr3** will be executed to modify the variable's value.

*) The above process continue until the **expr2** will becomes **FALSE**.

Examples:

1. for(i=1;i<=100;i++) /* Increasing */
   { body of loop }

i value is initialized to **1** and for every execution **i** value is incremented by **1** . So body of loop will execute **100** times.

2. for(i=100;i>0;i=i-2) /* Decreasing */
   { body of loop }

First value of **i** is initialized to **100** and for every execution **i** value is decremented by **2.** So body will execute **50** times.

The following program is used to calculate Factorial value for a given number **N.**

General formula : n**!** = 1 * 2 * 3 * 4* …….*n

To do so, any of the looping statements can be used. The initial value of the loop is 1, next increment is 1 and the summation.

```
/* To find the factorial of a number */
main( )
{
 int n, i, f=1;

 printf("\nEnter a number");
 scanf("%d",&n);
 for(i=1;i<=n;i++)
        f = f * i;
printf("\nFactorial of %d = %d ",n,f);
}
```

One more example, following is a program generates a *fibonacci series*. The series like as

0 1 1 2 3 5 8 13 . . .

Generally the number is obtained by summing up of previous two numbers. So, first initialize **0 , 1** to **a , b** respectively and find the number as c = a + b.

Next reassign the values of **a** and **b**, like b→ a, c→b and proceed the same way for N number of times.

```
        /* Fibonacci Sequence generation */
main( )
{
        int a=0,b=1,c,n,i;
```

```
        printf("\nHow many numbers :");
        scanf("%d",&n);

        printf("\nFibonacci Sequence \n");

        for(i=1;i<=n;i++)
          {
          c=a+b;
          printf("%d\t",c);
          a=b;   b=c;
         }
    }
```

Additional information about the for loop:

The for loop has three parts inside a set of parenthesis and each is separated by the semicolon (**;**).

The **expr1** may be placed before the for loop as

```
        i=10;
        for(;i<100;i++)
            {    }
```

We can have more than one statement in the place of **expr1,** which are separated by commas (**,**). For example

```
        for( a=4,i=0;i<10;i++)
          {      }
```

The **expr3** may also be placed in the body of the loop as .

```
        for(i=10;i<100;)
            {   i++;  }
```

If any expression in the **for** is missing, the semicolon must be placed.

```
        for(i=0;i<100;)
            {  i++; }     /* expr3  is   empty  in for  loop  /*
```

The following loop is used execute the statements indefinitely, because there is initialization, condition and modification.

```
        for( ; ; )
           {     }
```

More than one condition statement can be used in the **expr2** place of **for** loop as

```
        for(i=1;i<10&&j<20;i++)
            {    }
```

Nested for loop:

Just like a nested **if,** nested for loop is also possible. In the nested for loop, the statement part of the loop contains another for statement.

```
        for(expr1;expr2;expr3)  ——— O
                                ———
          for(expr4;expr5;expr6)      I

            {

              //  body of loop
```

Where **O** is an outer loop

I is an inner loop

Way of execution:

For every value of outer loop, the inner loop will execute no. of times. In the nested loop, the body of loop will be executed until both the expressions **expr2** and **expr5** becomes **FALSE**.

For example

```
for(i=1;i<=10;i++)
for(j=1;j<=5;j++)
        printf("\nIndia");
```

Here for every value of **i** the **j** loop execute the statement part **5** times. So totally the string **India** will be printed **50** times (10 * 5 = 50).

The following example shows a clear output for you about the nested loop.

```
/* Example for nested for loop  */
main( )
{
        int i,j;
        for(i=1;i<=10;i++)
          for(j=1;j<=5;j++)
                printf("\ni = %d      j  = %d   ",i,j);
}
```

Output:

```
i = 1      j  = 1
i = 1      j  = 2
.
i = 2      j  = 1
.
.
i = 10    j = 5
```

Here for every value of **i** the **j** loop executes **5** times.

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

How to Declare Two-dimensional Array?

Similar to single dimensional array and simple variable declaration, the two-dimensional array is declared as

```
Data type   variable  [ size1]  [ size2 ] ;
```

Where - **size 1** refers to no. of rows

- **size 2** refers to no. of columns

So totally we can store **size1 * size2** values in the **variable**.

Example: int a[20][10];

*) Here **a** is the two dimensional array variable and it has 2**0** rows and **10** columns. So totally we can store **200** (20 * 10=200) values in the variable **a**.

*) To refer any value we have to specify the row no. and column no.

Ex: a[2][5]

=> It refers to the element of 2nd row 5th column

Assigning Values

We can assign the values to the variable while declaration of it.

Ex : int a[2] [3] = { {1,2,3},
{4,5,6} };

*) Each row is separated by braces **{ }** and each element by commas.

*) Here two rows and three columns.

*) First row contains the values 1,2,3 and second row contains the values 4,5,6.

Addition of two matrix

```
#include <stdio.h>
 int main()
{
  int m, n, c, d, first[10][10], second[10][10], sum[10][10];

  printf("Enter the number of rows and columns of matrix\n");
  scanf("%d%d", &m, &n);
  printf("Enter the elements of first matrix\n");

  for (c = 0; c < m; c++)
    for (d = 0; d < n; d++)
      scanf("%d", &first[c][d]);

  printf("Enter the elements of second matrix\n");

  for (c = 0; c < m; c++)
    for (d = 0 ; d < n; d++)
      scanf("%d", &second[c][d]);

  printf("Sum of entered matrices:-\n");

  for (c = 0; c < m; c++) {
    for (d = 0 ; d < n; d++) {
      sum[c][d] = first[c][d] + second[c][d];
      printf("%d\t", sum[c][d]);
    }
    printf("\n");
  }

  return 0;}
```

(OR)

b. What is an array? Explain in detail the various types of arrays with example.

Array is a collection of elements or data items

*) All the elements must be same data type

*) and they are stored in consecutive memory locations

How to Declare Array variable?

Simple variables are declared as,

```
int a,b,c;    /* Simple variable Declaration */
```

data-type followed by list of variables.

Similarly, an array variable can also be as follows.

```
data type   variable   [ size ] ;
```

Where - **data type** is the type of data like int, char etc.,

- **variable** is the name of the array variable

- **size** is the maximum no. of elements to be stored in the array and the size must be an integer value.

Example for array declaration is

```
int   a [ 5 ] ;
```

*) **a** is the name of the array variable of type **int**eger *) In the variable **a,** we can store **5** integer values.

*) The memory allocation is as follows which assumes the starting address is **1000.** Each element of the array occupies **two** bytes because of integer data type.

Element	0	1	2	3	4
Memory Location	1000	1002	1004	1006	1008

How to refer the values of array variable?

In **C** the first element of array is stored at the location **0**. So the element can be accessed as follows:

- First element is referred as a[0] location 1000

- Second element is referred as a[1] location 1002

- Third element is referred as a[2] location 1004 etc.

*) Here **0,1,2, …** are called as subscript (or) index. So array is also called as **subscripted variable.**

*) The above array is called **single dimensional array**. Because to refer any data we need only one index.

Assigning the data into array:

Like a simple variable assignment, the values can be assigned to the array variable as shown in the example.

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

Here 1st element is stored at a[0]
2nd element is stored at a[1]
3rd element is stored at a[2]
4th element is stored at a[3] and
5th element is stored at a[4]

Following are some of the ways to assign values to array variables.

*) int a[10] = {20,30};

=> Here **10** memory locations are reserved for **a**. But we are using only **2**. So the remaining spaces (eight) are wasted.

*) int a[ ] = {10,20,30};

=> In this declaration the above problem has been solved. The size of array is adjusted automatically depending upon the no. of values assigned.

*) The **default value** of variable is **garbage value**.

*) The following declaration initializes all the values of array to **0**.

int a[100] = {0};

In this case all the locations are filled by the value **0**.

KARPAGAM ACADEMY OF HIGHER EDUCATION

(Deemed to be University)

(Established Under Section 3 of UGC Act 1956)

COIMBATORE – 641 021

INFORMATION TECHNOLOGY/COMPUTER TECHNOLOGY

First Semester

SECOND INTERNAL EXAMINATION - August 2018

PROGRAMMING FUNDAMENTALS USING C/C++

Class & Section: I B.Sc (IT) & I B.Sc CT **Duration: 2 hours**

Date & Session: 29.8.18 AN **Maximum: 50 marks**

Subj.Code: 18ITU101/18CTU101

PART- A (20 * 1= 20 Marks)

Answer ALL the Questions

1. ____ function joins two strings together
   a. strcat b. strcmp c. strcpy d. strlen
2. ____ function compares two strings identified by the arguments
   a. strcat **b. strcmp** c. strcpy d. strlen
3. strcmp function returns the value_____ if the arguments are equal
   a. zero b. one c. two d. three
4. _____ function assigns the contents of one string to another
   a. strcat b. strcmp **c. strcpy** d. strlen
5. ._____ function counts and returns the number of characters in a string
   a. strcat b. strcmp c. strcpy **d. strlen**
6. Individual values in array is referred as _____
   a. subscript c. subelements
   b. elements d. pointers
7. _____ functions has to be developed by the user at the time of developing a program
   a. user defined b. built in c. subroutines d. structure
8. _____ header file should be declared to call input and output function
   a. stdio.h b. stdlib.h c. conio.h d. math.h
9. Header file stdio.h calls _____ function
   a. input/output b. math c. character d. sqrt
10. In function declaration if the return type is not specified , it returns _____ by default
    a. integer b. character c. float d. long
11. _____ method is for packaging data of different data type
    a. structure b. union c. pointer d. function
12. _____ keyword is used to declare structure
    a. bit **b. struct** c. union d. array
13. Fields within structure are called _____
    a. data b. variable **c. member** d. subelements
14. Linking member and a variable in structure is established using ____operator
    a. * b. & **c. dot** d. >
15. Structure must be declared _____ if it is to be initialized inside function
    a. intern b. extern **c. static** d. register
16. _____ is convenient tool for handling group of logically related data items
    a. union **b. structure** c. bitfields d. array

17. All members of _____ use the same storage location in memory
    a. union b. structure c. bitfields d. array
18. _____ is called the member operator
    a. * asterisks **c. . Period**
    b. ; semicolon d. : colon
19. Each member of a structure must be declared independently for its _____ in a separate statement inside the structure template
    a. size and name c. height and length
    b. name and type d. scope and life
20. Convenient way of representing the details of all students in a class is ____
    a. array of structures c. array of names
    b. two dimensional array d. structure with arrays

PART- B (3 * 2= 6 Marks)
Answer ALL the Questions

21. Write notes on i) Call by value ii) Call by reference of functions
22. What is a function? How will you define a function?
23. How the members of structure are accessed? Write example

PART C (3 * 8 = 24 Marks)
Answer ALL the Questions

24. a. Explain in detail about String functions with syntax and example.

(OR)

b. Explain the Array of Structures with example.

25. a. Explain the various types of user-defined functions with example.

(OR)

b. How will you declare, initialize and access a structure? Write a program to calculate net pay of an employee using structure.

26. a. How will you declare, initialize and access a union? Explain in detail with example.

(OR)

b. Write a program to swap two numbers using pointers.

KARPAGAM ACADEMY OF HIGHER EDUCATION

(Deemed to be University)

(Established Under Section 3 of UGC Act 1956)

COIMBATORE – 641 021

INFORMATION TECHNOLOGY/COMPUTER TECHNOLOGY

First Semester

SECOND INTERNAL EXAMINATION - August 2018

PROGRAMMING FUNDAMENTALS USING C/C++

Class & Section: I B.Sc (IT) & I B.Sc CT **Duration: 2 hours**

Date & Session: 29.8.18 AN **Maximum: 50 marks**

Subj.Code: 18ITU101/18CTU101

PART- A (20 * 1= 20 Marks)

Answer ALL the Questions

1. ____ function joins two strings together
   - **a. strcat**
   - b. strcmp
   - c. strcpy
   - d. strlen
2. ____ function compares two strings identified by the arguments
   - a. strcat
   - **b. strcmp**
   - c. strcpy
   - d. strlen
3. strcmp function returns the value_____ if the arguments are equal
   - **a. zero**
   - b. one
   - c. two
   - d. three
4. _____ function assigns the contents of one string to another
   - a. strcat
   - b. strcmp
   - **c. strcpy**
   - d. strlen
5. ._____ function counts and returns the number of characters in a string
   - a. strcat
   - b. strcmp
   - c. strcpy
   - **d. strlen**
6. Individual values in array is referred as _____
   - a. subscript
   - **b. elements**
   - c. subelements
   - d. pointers
7. _____ functions has to be developed by the user at the time of developing a program
   - **a. user defined**
   - b. built in

c. subroutines
d. structure

8. _____ header file should be declared to call input and output function
   a. stdio.h
   b. stdlib.h
   c. conio.h
   d. math.h
9. Header file stdio.h calls _____ function
   a. input/output
   b. math
   c. character
   d. sqrt
10. In function declaration if the return type is not specified , it returns _____ by default
    a. integer
    b. character
    c. float
    d. long
11. _____ method is for packaging data of different data type
    a. structure
    b. union
    c. pointer
    d. function
12. _____ keyword is used to declare structure
    a. bit
    b. struct
    c. union
    d. array
13. Fields within structure are called _____
    a. data
    b. variable
    c. member
    d. subelements
14. Linking member and a variable in structure is established using ____operator
    a. *
    b. &
    c. dot
    d. >
15. Structure must be declared _____ if it is to be initialized inside function
    a. intern
    b. extern
    c. static
    d. register
16. _____ is convenient tool for handling group of logically related data items
    a. union
    b. structure
    c. bitfields
    d. array
17. All members of _____ use the same storage location in memory

a. union
b. structure
c. bitfields
d. array

18. _____ is called the member operator
    a. * asterisks
    b. ; semicolon
    c. . Period
    d. : colon
19. Each member of a structure must be declared independently for its _____ in a separate statement inside the structure template
    a. size and name
    b. name and type
    c. height and length
    d. scope and life

20. Convenient way of representing the details of all students in a class is ____

a. array of structures
b. two dimensional array
c. array of names
structure with arrays

PART- B (3 * 2= 6 Marks)
Answer ALL the Questions

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

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

23. How the members of structure are accessed? Write example

PART C (3 * 8 = 24 Marks)
Answer ALL the Questions

24. a. Explain in detail about String functions with syntax and example.

Library Functions in String:

The operations like, copying a string, joining of two strings, extracting a portion of the string, determining the length of a string etc. cannot be done with arithmetic operators. String based library functions are used to perform this type of operations and they are located in the header file **<string.h>.**

Four important string based library functions are

1. strlen( ) - To find the length of a string
2. strcpy( ) - To copy one string into another
3. strcat( ) - To join strings
4. strcmp( ) - To compare two strings.

Function	Use
strlen	Finds length of a string
strlwr	Converts a string to lowercase
strupr	Converts a string to uppercase
strcat	Appends one string at the end of another
strncat	Appends first n characters of a string at the end of another
strcpy	Copies a string into another
strncpy	Copies first n characters of one string into another
strcmp	Compares two strings
strncmp	Compares first n characters of two strings
strcmpi	Compares two strings without regard to case ("i" denotes that this function ignores case)
stricmp	Compares two strings without regard to case (identical to strcmpi)
strnicmp	Compares first n characters of two strings without regard to case
strdup	Duplicates a string
strchr	Finds first occurrence of a given character in a string
strrchr	Finds last occurrence of a given character in a string
strstr	Finds first occurrence of a given string in another string
strset	Sets all characters of string to a given character
strnset	Sets first n characters of a string to a given character
strrev	Reverses string

1. Length of the string - strlen( ):

The function, **strlen( )** is used to find the length of the given string. The general format is as follows.

```
int     strlen ( str );
```

Where **str** is a string variable and it returns number of characters present in the given string.

Example:

```
char   str[10] = "Karthi";
       len = strlen(s)
```

=> It returns length of the string as **6** and it is stored in the variable **len**.

```
char  s[10] = "Welcome\0";
```

len = strlen(s);

=> This example returns length as **7,** because it will not count NULL character ('\0') as a character of the string .

len=strlen("AB**bbb**CD ");

=> where **b** is a blank space. In this case the blank space is also treated as a character. So, length of this string is **7**. (Including blank space)

The Null character (\0) is not a countable character in the string

2 Assigning the string - strcpy ( ) :

The function **strcpy( )** is used to copy the content of one string into another. We can't use the **assignment operator** (=) to assign a string to the variable. By using this function only we can perform assignment operation on string.

```
char s[15] ;
s  = "Man";   /* This is not possible in C */
```

The general format is as follows.

```
strcpy( s1, s2);
```

Where **s1** and **s2** are string variable

Here **s2** is the source string and **s1** is the destination string. After the execution of this function content of **s2** is copied into **s1** and finally both **s1** and **s2** contains the same values.

Example :

1. char s1[ ] = "Karthi";
   char s2[ ] = "Good";
   strcpy(s1,s2);

   => After the execution the content of the string **s2** is copied into string **s1**. Therefor the string **"Karthi"** is replaced with the string **"Good"**. Now both **s1** and **s2** contains the string **"Good"**.

2. char s1[10];
   char s2[10] = "Karthi";

strcpy(s1,s2);

=> Here also the content of **s2** is copied into **s1** and both

contains the string as **Karthi**.

3 Joining Strings - strcat( ) :

The function **strcat( )** is used for joining two strings. The general format is as follows

```
strcat(s1 , s2 );
```

Where **s1** and **s2** are string variables.

Here the content of **s2** is appended (or) joined to the contents of **s1**. After the execution **s1** now contains its own content, followed by the contents of **s2** and **s2** retains the same.

Example:

1. char s1[10] = “Good” , s2 [10] = “Morning”;

   strcat (s1,s2);

   => After the execution of this function, **s1** contains

   "**GoodMorning"** and **s2** contains "**Morning"**.

2. char s1[10] = “ ” , s2 [10] = “Welcome”;

   strcat (s1,s2);

   => After execution, both **s1, s2** has "**Welcome",** because

   first variable **s1** does not have any character in it.

4 To Compare - strcmp( ) :

This function is used to compare two strings. To compare any numeric values we use relational operators, by using these operators we can’t compare strings. The **strcmp( )** performs the function of comparison. The general format is as follows.

```
int    strcmp(s1 , s2 );
```

Where **s1** and **s2** are string variables.

This function returns any one of three possible results.

*) Result is **0** when both strings are equal. (s1 = s2)

*) Result is **Positive** value, if **s1** is greater than **s2** (s1>s2)

*) Result is **Negative** if **s1** is less than **s2**. (s1<s2)

Note:

The result of comparison is obtained by calculating the difference between the ASCII value of the corresponding characters. The comparison is made by checking the corresponding characters (ASCII values) one by one between two strings.

The first character of s1 is compared with the first character of s2. If they are equal, the process passes on to the next character on the string until they meet with mismatch or no more character to process.

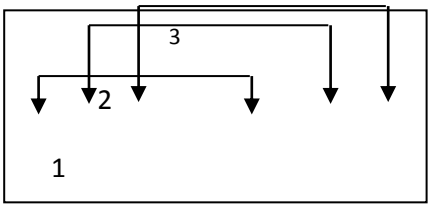
The process of comparison is terminated if any mismatching occurs or end of string is reached.

Example:

1. char s1[5] = “ABC”;

   char s2[5] = “ABC”;

   strcmp(s1,s2);



=> This function returns **0** as a result, because both **s1** and **s2** contains the same characters. The characters of **s1** is equal to the character of **s2**. So, the result is **0**. (ASCII difference of these characters).

2. char s1[5] = “ABC”;

   char s2[5] = “abc”;

   strcmp(s1,s2);

=>This function returns -**32** as a result, as ASCII difference between **A** and **a** is -**32** (65 – 97). ASCII value of **'A'** is **65** and '**a'** is **97.**

3. char s1[5] = “ABz”;

   char s2[5] = “ABC”;

   strcmp(s1,s2);

=> This function returns **55** as a result. Because ASCII difference between **z** and **C** is **55**(122 – 67). ASCII value of **'z'** is **122** and **'c'** is **67.**

Here is a program to illustrate, how to read a string and find out its length without using the library function.

```
/* Implementing strlen( ) function */

#include <string.h>
```

```
main( )
{
 int i= 0;
 char str[25];
 printf("\nEnter a string    : ");
 gets(str);
 printf("\nYour given string : ");
 while(str[i])
   printf("%c",str[i++]);
 printf("\nLength of string  : %d ",i);
}
```

Output:

```
 Enter a string    : karthi
 Your given string : karthi
 Length of string  : 6
```

/*Check whether the given string is palindrome or not*/

```
#include <string.h>
main( )
{
        int i,l,poly=1;
        char s[50];
        printf("\nEnter a string    : ");
        gets(s);
        printf("\nGiven string  is  : %s",s);
        l=strlen(s);
        printf("\nLength of string  :  %d\n",l);
        l = l-1;
```

```
        for(i=0;i<=l;i++)
            if (s[i]!=s[l-i])
                poly=0;
        if (poly = =1)
                printf("\n'%s'  is polindrome",s);
        else
                printf("\n'%s'  is not polindrome",s);
    }
```

strrev( ) => This function is used to reverse the string

```
char s[ ] =”Hello”;
strrev(s);
```

=>Now the content of the string **s** is reversed.

A program to find out whether the given string is palindrome or not using the library function.

```
/* Palindrome checking using library function */
main()
{
        char s1[15],s2[15];
clrscr();
        printf("\nEnter a string : ");
        scanf("%s",s1);
        strcpy(s2,s1);
        strrev(s2);
        if (strcmp(s1,s2)==0)
           printf("\nGiven strnig is palindrome ");
        else
           printf("\nGiven string is not palindrome ");
        getch();  }
```

(OR)

b. Explain the Array of Structures with example.

d. Array of structures:

e. 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

f.
```
struct  emp e[100];
```

g. 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

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

i. 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);
  }      }
```

j. 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)

k.

```
/* 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
```

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

25. a. Explain the various types of user-defined functions with example.

User defined functions – Design your own:

The user can write a function according to their wish and requirements and this type of function is called as user defined function. It is just like designing a dress depending upon one's taste. So if you are not satisfied with the readymade, design your own.

The purpose of having a function in a program is to reduce the size of the program and in some cases this can also be achieved by using the looping statements. The following is a program and how the same is reduced is illustrated.

```
main()
{
        printf("\nHello");
        printf("\nGood morning");    Set 1
        printf("\nHello");          Set 2
        printf("\nGood morning");
        printf("\nHello");
        printf("\nGood morning");
        printf("\nHello");
        printf("\nGood morning");
        printf("\nHello");
        printf("\nGood morning");
    printf("\nHello");
        printf("\nGood morning");
}
```

Can we reduce the size of the above program? Yes. We can. The following is a revised version of the program using for looping statement.

```
/*  Minimized program using for loop */
main()
{       int i;
        for(i=1;i<=3;i++)
        {       printf("\nHello");
                printf("\nGood morning");
        }       }
```

Suppose the repetition occurs in different part of the program instead of continuous one, using looping is not a solution. Here comes the function. Yes. Keep the repeated set of statements in a separate part of the program (some time called as sub-program). Whenever these repeated state[illegible]d, the sub-[illegible]voked and utilized.

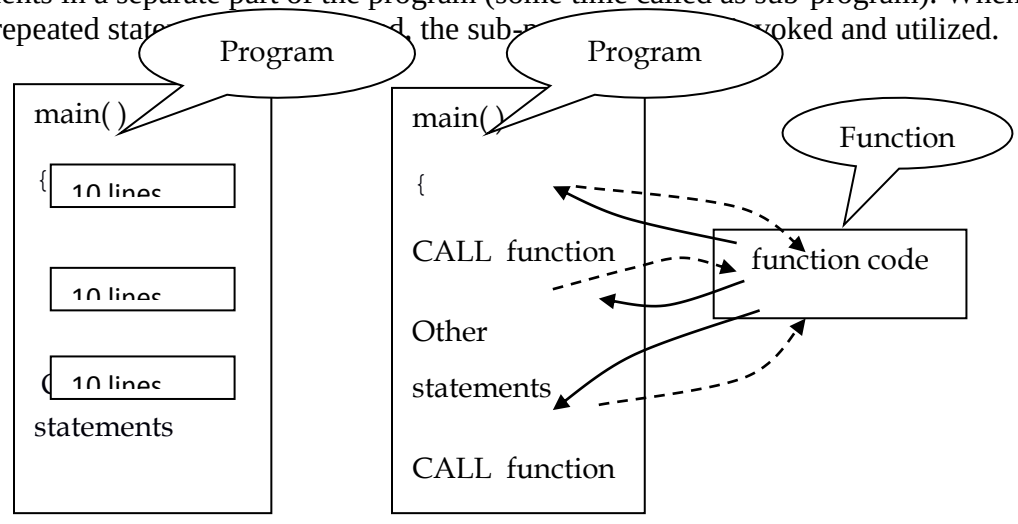
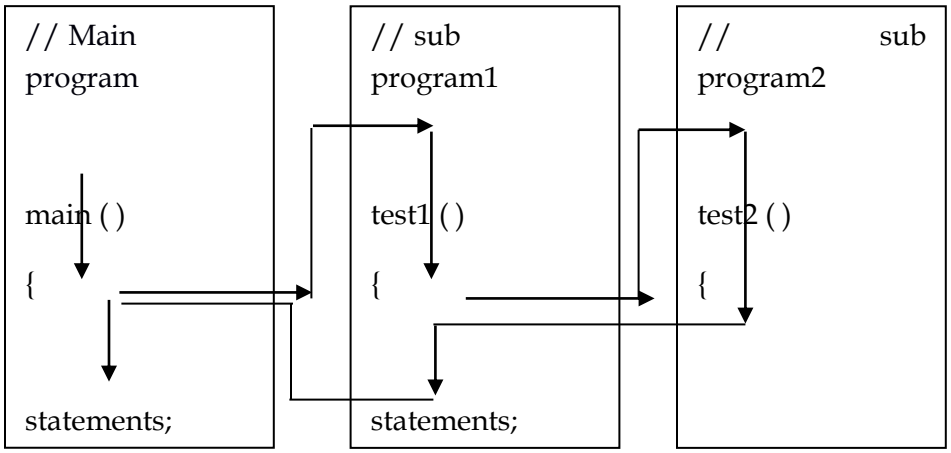


Figure a Figure b

Figure (a) is a program with repeated code in three places and the aim of all the 10 lines are same. In figure (b) the repeated statements are available as a sub-program / function. This function can be invoked/called wherever the repeated code is necessary.

In the figure (a) the repeated code is 10 lines, totally it occupies 30 lines, because it is in 3 places (3 * 10 = 30). But in the figure (b) instead of 10 lines, only one instruction (CALL) is used to invoke the sub program. Totally 30 lines of the figure (a) is reduced into 3 lines and 10 lines in the function. This function can be invoked any number of times at any place.

The following diagram is illustrating how the functions are being invoked and processed with multiple functions.



In the above diagram there is two sub programs namely test1 and test2.

- First we know the main( ) function starts its execution and it calls the user defined function test1( )
- Before starting process, the status of main( ) is stored into stack and execution continuous in test1( )
- The function test1( ), in turn calls another function test2( )
- As in the previous case status of test1( ) is pushed in to the stack.
- Now stack contains status both test1( ) function and main ( )
- The execution continuous in test2( )
- After completion of test2( ), the control is returned to the test1( ) and continues the execution until the end of the function
- When test1( ) is completed, the control returns to main( ) program and once again the process is resumed.

General format for function declaration:

```
Return-type function-name(arg1,arg2...)
     {

       Local variable declaration
```

Here

- ⇨ Return-type is type of data returned by the function
- ⇨ Function-name is the name of the function
- ⇨ arg1 , arg2 … are parameters of the function

Look the example for a function declaration

```
int swap(int a,  int b)
{
    // Body of the function;
}
```

The first int is the type of data to be returned by the function swap( ) and a, b are the arguments to the function.

(OR)

b. How will you declare, initialize and access a structure? Write a program to calculate net pay of an employee using structure.

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

```
struct   <tag>
   {
      member-1;
      member-2;
```

```
struct  [<tag>]
   {
     member-1;
      member-2;
```

- ⇨ **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);
}
```

26. a. How will you declare, initialize and access a union? Explain in detail with example.

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;
```

```
union
 {
    char
name[15];
    int   rollno;
```

he size bytes **(15+2+4)**. But it is not arger m ered for allocation. No inde bers wil

riable *

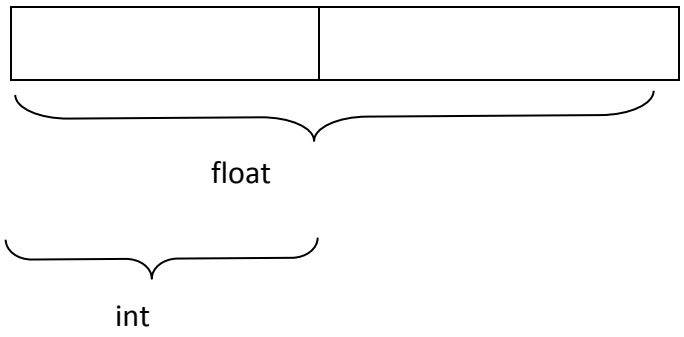
```
{union
        { char name[15];
         int  rollno;
         float marks;
        } s;
```

```
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

(OR)

b. Write a program to swap two numbers using pointers.

AIM

To write a program that swaps two numbers using pointers and macros.

ALGORITHM

STEP 1 : Start the program.

STEP 2 : Declare the necessary variables.

STEP 3 : Get the input for first number.

STEP 4 : Get the input for second number.

STEP 5 : Call the function swap() by passing the address of the two numbers as arguments.

STEP 6 : Swap the two number using a temporary variable temp.

STEP 7 : Print the swapped numbers.

STEP 8 : Stop the program.

SWAPPING TWO NUMBERS

```
#include<stdio.h>
#include<conio.h>
void SWAP(intx,int y)
{
intt;t=x;x=y;y=t;
printf("\n Aterswaping the value of a= %d and b= %d",x,y);
}
void main ()
{
int a1,b1,x,y,*a,*b,temp,op,j;
clrscr();
do
 {
printf("\n\t options");
printf("\n 1.swaping two number using macros");
printf("\n 2.swaping two number using pointers");
printf("\n Enter your option(1/2):");
scanf("%d",&op);
switch(op)
  {
```

```
case 1:
printf("\nEnter two number: ");
scanf("%d%d",&a1,&b1);
printf("\n Before swaping the value of a= %d and b =%d",a1,b1);
SWAP(a1,b1);
break;
case 2:
printf("\nEnter the vale of x and y: ");
scanf("%d%d",&x,&y);
printf("Before swaping \n x=%d \n y=%d \n",x,y);
   a=&x;
   b=&y;
temp=*b;
   *b=*a;
   *a=temp;
printf("After swaping \n x=%d \n y=%d \n",x,y);
break;
  }
 }
while(op<2);
getch();
}
```

OUTPUT:

```
options
1.swaping two number using macros
2.swaping two number using pointers
 Enter your option(1/2):1
Enter two number: 15
10
 Before swaping the value of a= 15 and b =10
Aterswaping the value of a= 10 and b= 15
options
1.swaping two number using macros
2.swaping two number using pointers
```

Enter your option(1/2):2

Enter the vale of x and y: 15
10
Before swaping
x=15
y=10
After swaping
x=10
y=15

Result:

The above program has been executed successfully and the output is verified.

KARPAGAM ACADEMY OF HIGHER EDUCATION

(Deemed to be University)

(Established Under Section 3 of UGC Act 1956)

COIMBATORE – 641 021

INFORMATION TECHNOLOGY/COMPUTER TECHNOLOGY

First Semester

THIRD INTERNAL EXAMINATION - October 2018

PROGRAMMING FUNDAMENTALS USING C/C++

Class & Section: I B.Sc (IT) & I B.Sc CT **Duration: 2 hours**

Date & Session: 8.10.18 AN **Maximum: 50 marks**

Subj.Code: 18ITU101/18CTU101

PART- A (20 * 1= 20 Marks)

Answer ALL the Questions

1. In ____ mode the existing file is opened for reading only
   a. r b. w c. a d. f
2. In _____ mode the file can be opened for writing only
   a. r **b. w** c. a d. f
3. In _____ mode the file can be opened for appending data to it
   a. r b. w **c. a** d. f
4. The getc function will return an _____ ,when end of the file has been reached
   a. BOF **b. EOF** c. SOF d. FOF
5. _____ is a parameter supplied to a program when the program is invoked
   a. argument **c. command line argument**
   b. parameter d. values
6. Command line argument is supplied to the program when it is _____
   a. invoked b. developed c. compiled d. stored
7. The file mode _____ is used to read and append some data into an existing file from end of the file
   a. "r" b. "r-" c. "r+" d. all
8. A _____ file is a collection of ASCII characters, with end of line markers and end of file markers
   a. program b. binary c. image **d. text**
9. fclose() function is used to close
   a. editor b. program c. all **d. file**
10. File mode must be specified while _____
    a. opening a file c. writing a file
    b. reading a file d. closing a file
11. Objects can be_____
    a. created c. permanent
    b. created & destroyed d. temporary
12. _____helps the programmer to build secure programs
    a. Dynamic binding c. Data building
    b. Data hiding d. message passing

13. _____techniques for communication between objects makes the interface descriptions with external systems much simpler
    a. message passing c. Encapsulation
    b. Data binding d. Data passing
14. _____are extensively used for handling class objects
    a. overloaded functions c. objects
    b. methods d. messages
15. _____refers to the act of representing essential features without including the background details or explanations
    a. encapsulation c. Dynamic binding
    b. inheritance **d. Abstraction**
16. Attributes are sometimes called_____
    a. data members c. messages
    b. methods d. functions
17. The functions operate on the datas are called_____
    a. methods c. messages
    b. data members d. classes
18. _____is the process by which objects of one class acquire the properties of objects of another class
    a. polymorphism **c.** data binding
    b. encapsulation **d. Inheritance**
19. _____ is used to read a number of items from the file stream using format
    a. printf() b. scanf() c. fprintf() **d. fscanf()**
20. A _____ constructor is used to declare and initialize an object from another object
    a. Default **c.** multiple
    b. copy **d.** parameterized

PART- B (3 * 2= 6 Marks)
Answer ALL the Questions

21. What is memory allocation in C++? Define its types.
22. How will you open a file? Give example
23. Write about new and delete operators with example.

PART C (3 * 8 = 24 Marks)
Answer ALL the Questions

24. a. Explain reading and writing text files.

(OR)

b. Write a C program to copy the content of a file into another.

25. a. List the basic concepts of C++ and explain it

(OR)

b. Explain Random access file.

26. a. Difference between procedure-oriented and object-oriented programming

(OR)

b. Define class. How will you create class and explain with example.

KARPAGAM ACADEMY OF HIGHER EDUCATION

(Deemed to be University)

(Established Under Section 3 of UGC Act 1956)

COIMBATORE – 641 021

INFORMATION TECHNOLOGY/COMPUTER TECHNOLOGY

First Semester

THIRD INTERNAL EXAMINATION - October 2018

PROGRAMMING FUNDAMENTALS USING C/C++

Class & Section: I B.Sc (IT) & I B.Sc CT **Duration: 2 hours**

Date & Session: **Maximum: 50 marks**

Subj.Code: 18ITU101/18CTU101

PART- A (20 * 1= 20 Marks)

Answer ALL the Questions

1. In ____ mode the existing file is opened for reading only
   - **a. r**
   - b. w
   - c. a
   - d. f
2. In _____ mode the file can be opened for writing only
   - a. r
   - **b. w**
   - c. a
   - d. f
3. In _____ mode the file can be opened for appending data to it
   - a. r
   - b. w
   - **c. a**
   - d. f
4. The getc function will return an _____ ,when end of the file has been reached
   - a. BOF
   - **b. EOF**
   - c. SOF
   - d. FOF
5. _____ is a parameter supplied to a program when the program is invoked
   - a. argument
   - b. parameter
   - **c. command line argument**
   - d. values
6. Command line argument is supplied to the program when it is _____
   - **a. invoked**
   - b. developed
   - c. compiled
   - d. stored
7. The file mode _____ is used to read and append some data into an existing file from end of the file

a. "r"
b. "r-"
c. "r+"
d. all

8. A _____ file is a collection of ASCII characters, with end of line markers and end of file markers
   a. program
   b. binary
   c. image
   d. text
9. fclose() function is used to close
   a. editor
   b. program
   c. all
   d. file
10. File mode must be specified while _____
   a. opening a file
   b. reading a file
   c. writing a file
   d. closing a file
11. Objects can be_____
   a. created
   b. created & destroyed
   c. permanent
   d. temporary
12. _____helps the programmer to build secure programs
   a. Dynamic binding
   b. Data hiding
   c. Data building
   d. message passing
13. _____techniques for communication between objects makes the interface descriptions with external systems much simpler
   a. message passing
   b. Data binding
   c. Encapsulation
   d. Data passing
14. _____are extensively used for handling class objects
   a. overloaded functions
   b. methods
   c. objects
   d. messages
15. _____refers to the act of representing essential features without including the background details or explanations
   a. encapsulation
   b. inheritance
   c. Dynamic binding
   d. Abstraction
16. Attributes are sometimes called_____
   a. data members

b. methods
c. messages
d. functions

17. The functions operate on the datas are called_____
    a. methods
    b. data members
    c. messages
    d. classes
18. _____is the process by which objects of one class acquire the properties of objects of another class
    a. polymorphism
    b. encapsulation
    c. data binding
    d. Inheritance
19. _____ is used to read a number of items from the file stream using format
    a. printf()
    b. scanf()
    c. fprintf()
    d. fscanf()
20. A _____ constructor is used to declare and initialize an object from another object
    a. Default
    b. copy
    c. multiple
    d. parameterized

PART- B (3 * 2= 6 Marks)
Answer ALL the Questions

21. What is memory allocation in C++? Define its types.

There are two ways that memory gets allocated for data storage:

1. Compile Time (or static) Allocation
   - Memory for named variables is allocated by the compiler
   - Exact size and type of storage must be known at compile time
   - For standard array declarations, this is why the size has to be constant
2. Dynamic Memory Allocation
   - Memory allocated "on the fly" during run time
   - dynamically allocated space usually placed in a program segment known as the *heap* or the *free store*
   - Exact amount of space or number of items does not have to be known by the compiler in advance.
   - For dynamic memory allocation, pointers are crucial

22. How will you open a file? Give example

The purpose of using a file may be to read or write data. To do any of the two functions first we must open the file. The file can be opened by using the library function **fopen( )** and its format

```
FILE *fp  =  fopen(filename, mode);
```

is

Where filename **is the name of the file to be opened and** mode **is the purpose of opening the file like read / write / append.**
If the file exists, it is opened and the **fopen( )** function **returns the starting address** of t he file. If **fail** on opening the result is **NULL**.

For example

fopen(“stu.dat”,”r”);

=>”**stu.dat”** is the name of the file and “**r”** is mode of opening the file (“**r”** indicates the read mode). If success, it returns the starting address of **stu.dat** file.

The file may be opened in any one of the following modes:-

	Description
w	
r	Open the available file for reading
a	Open the file for appending
w+	Create the new file for both operations (read & write)
r+	Open the available file for both operations (read & write)
a+	Open the available file for both (read & write) operation with append

23. Write about new and delete operators with example.

new and delete operators are provided by C++ for runtime memory management. They are used for dynamic allocation and freeing of memory while a program is running.
- The new operator allocates memory and returns a pointer to the start of it. The delete operator frees memory previously allocated using new.
- The general form of using them is :

```
p_var = new type;

delete p_var;
```

PART C (3 * 8 = 24 Marks)
Answer ALL the Questions

24. a. Explain reading and writing text files.
to read the characters or write the character into a file. The library functions **fgetc( ), fputc( ), getc( ), putc( )** are used for the above said operations. The following discussions will give an idea about these functions.

getc( ) - To read a character from the file and adjust the file pointer

to the next character of the file automatically.

fgetc( ) - to read a character from a file.

putc( ) - to write a character into the file.

fputc( ) - to write a character into the file.

The general format of the getc( ) and fgetc( ) function is as follows.

```
char    getc( file-pointer);
```

These functions reads character one by one from the file and the file pointer will be automatically adjusted to the next location in the file. By this way we can read all the characters from the file, sequentially. The characters are fetched from the file and it can be displayed as like below.

```
ch = fgetc(fp);   /* Getting character from file */

putch(ch);                /* Displaying it in the screen */
```

Note: *File pointer will be automatically adjusted to the next location*

feof( ) - To check end of file

During the processing of the file, the end of file can be realized by the value the file pointer returns. The file pointer returns **0** if the end of file has been reached.

The end of file can be checked by using the library function **feof( )**. This function returns **TRUE** if the file reaches the end, otherwise **FALSE**. Using **feof( )** the process to be carried out in the file can be continued. For example, the statement

```
FILE  *fp=fopen("test.dat","r");

result =  feof(fp);
```

If the file reaches the end, the value in the **result** is **TRUE** otherwise the value in the **result** is **FALSE**. We can use a while loop for reading no. of characters as follows.

```
while( !feof( fp ) )

{

  /* Statements   */

}
```

Here the statement part will be executed until the end of file. Using a predefined constant, **EOF** can also check the end of character in the file. The following program will give an idea about how to read and check the end of file character.

```
/* Reading the characters from the file */
```

```
#include <stdio.h>
main( )
{   FILE *fp;
  char ch;
   fp=fopen("test.txt","r");
  if (fp= =NULL)
  {      printf("\nNo such file this name");
     exit(0);   }
 while (!feof(fp))
 {        ch = getc(fp);
          putch(ch);
  } }
```

In this program the file **test.txt** has been opened for reading the data. Using the function **getc(fp)** the characters are read, and displayed in the screen using **putch( )**. The function **feof( )** is used to check the occurrence of end of file. The process will be terminated when end of file is reached.

The general format of the **putc( )** and **fputc( )** is as follows.

```
putc  ( character , file-pointer);
fputc ( character , file-pointer);
```

where the **character** denotes the character to be written and **file-pointer** indicating the file which is receiving the character. These functions are used to write characters into the specified file.

For example

```
fputc('a', fp);
     /* character 'a' is written in the file */
```

The following example illustrates the **putc( )** function.

```
/* Creating New file by reading characters */

#include <stdio.h>

main( )

{ FILE *fp;

 char ch;

 fp=fopen("test.txt","w");

 if (fp= =NULL)

   {  printf("\nNo such file");

      exit(0);          }

  while((ch=getch( )) != 'z' )

  {   putc(ch,fp);

       putch(ch);   }

fclose(fp);     }
```

While executing this program the user can type any no. of characters and all the characters will be stored in the file **test.txt**. The process will be continued until the key **z is** pressed. Instead of **z** any other character can also be used to denote the end.

For example the user can give the following text, and the termination is with the character **z.**

No gain without pain z

Using the functions **getc( )** and **putc( ),** content of one file can be copied into another file as in the following example program.

```
/* Copy the content of one file to another  */
#include <stdio.h>
main( )
{ FILE *fps,*fpd;    /* source and destination */
 char ch, sfile[15], dfile[15];
 clrscr( );
 printf("\nSource file        : ");
 scanf("%s",sfile);
```

```
        printf("\nDestination file : ");
        scanf("%s",dfile);
        fps = fopen(sfile,"r"); /*source file should be opened for read  */
        fpd = fopen(dfile,"w"); /* This file should be opened for write  */
        if  (fps= =NULL)
        {     printf("\nNo source file in the directory");
              exit(0);       }
        while(!feof(fps))
        {         ch = getc(fps);
                  putc(ch,fpd);
        }
        fcloseall( );    /* Closes  all the opened files */
  }
```

Output:

```
                  Source  file       :   ek.c
                  Destination  file   :   mahi.c
```

After the execution, the content of **ek.c** is copied into the file **mahi.c**

(OR)

b. Write a C program to copy the content of a file into another.

```
/* Copy the content of one file to another  */
        #include <stdio.h>
        main( )
        { FILE *fps,*fpd;    /* source and destination */
         char ch, sfile[15], dfile[15];
         clrscr( );
         printf("\nSource file        : ");
         scanf("%s",sfile);
         printf("\nDestination file : ");
         scanf("%s",dfile);
         fps = fopen(sfile,"r"); /*source file should be opened for read  */
         fpd = fopen(dfile,"w"); /* This file should be opened for write  */
         if  (fps= =NULL)
         {     printf("\nNo source file in the directory");
               exit(0);       }
         while(!feof(fps))
         {         ch = getc(fps);
                   putc(ch,fpd);
         }
         fcloseall( );    /* Closes  all the opened files */
  }
```

Output:

```
                  Source  file       :   ek.c
                  Destination  file   :   mahi.c
```

25. a. List the types of inheritance. Explain multi-level inheritance with program.

Inheritance

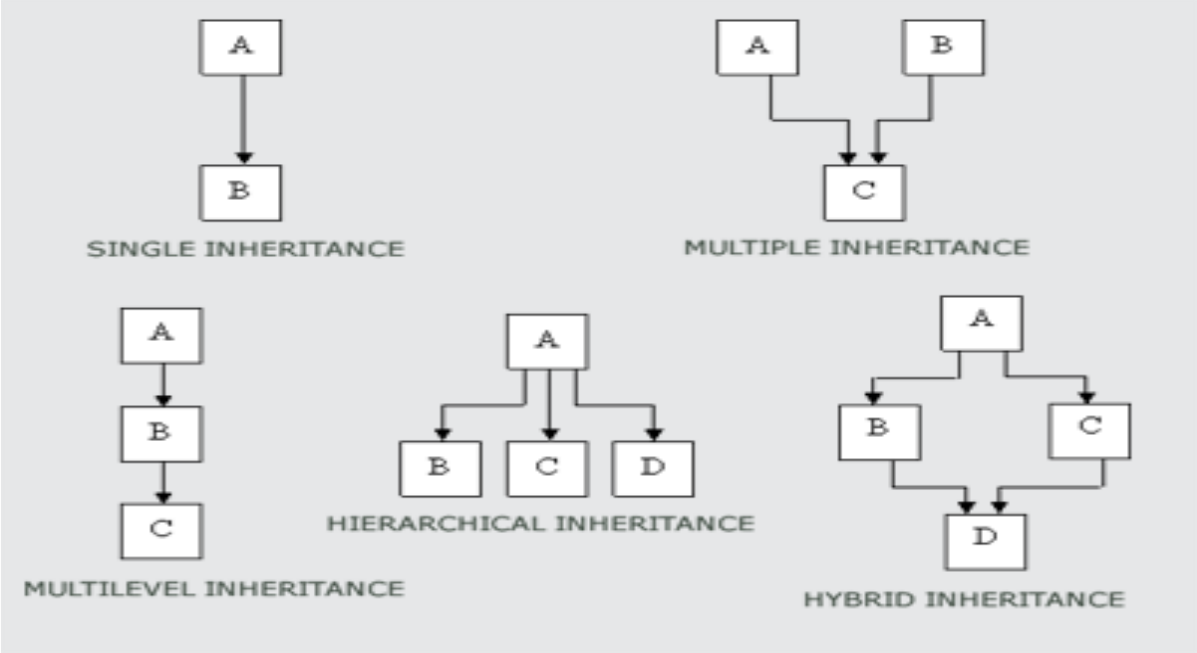
The mechanism that allows us to extend the definition of a class without making any physical changes to the existing class is inheritance.

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.

Multilevel Inheritance

Multilevel Inheritance is a method where a derived class is derived from another derived class.

```
#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;
     }
    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

(OR)

b. Discuss Macros with example program.

26. a. What is overloading. Describe with syntax and example for method overloading.

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
```

(OR)

b. Explain copy constructors with suitable example.

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

```
#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

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

```
#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

Reg. No.....

[17CTU203]

KARPAGAM ACADEMY OF HIGHER EDUCATION

(Deemed to be University Established Under Section 3 of UGC Act 1956)
Pollachi Main Road, Eachanari Post, Coimbatore – 641 021.
(For the candidates admitted from 2017 onwards)

B.Sc., DEGREE EXAMINATION, APRIL 2018

Second Semester

COMPUTER TECHNOLOGY

DATABASE PROGRAMMING WITH ORACLE (SQL AND PL/SQL)

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 meant by DBMS.
22. Write the basic components of Data Model.
23. Define SQL.
24. Define Exception
25. Define Package.

PART C (5 x 6 = 30 Marks)
Answer ALL the Questions

26. a. Explain about structure of DBMS.

Or

b. Explain about Network data model.

27. a. Discuss about Relational Algebras basic operation.

Or

b. Explain about Attribute , Domain, Tuple in Relational Database

28. a. Explain about Data Definition and Data Manipulation in SQL.

Or

b. Explain about SQL Join.

29. a. Explain about PL/SQL Block Structure.

Or

b. Explain about trigger with example.

30. a. Explain about Boyce Codd normal form with an example.

Or

b. Explain about Fourth and Fifth normal form with example.

Scanned by CamScanner