NEHRU COLLEGE OF ENGINEERING AND RESEARCH CENTRE
(NAAC Accredited)
(Approved by AICTE, Affiliated to APJ Abdul Kalam Technological University, Kerala)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

COURSE MATERIALS

CS100 COMPUTER PROGRAMMING

VISION OF THE INSTITUTION

To mould true citizens who are millennium leaders and catalysts of change through excellence in education.

MISSION OF THE INSTITUTION

NCERC is committed to transform itself into a center of excellence in Learning and Research in Engineering and Frontier Technology and to impart quality education to mould technically competent citizens with moral integrity, social commitment and ethical values.

We intend to facilitate our students to assimilate the latest technological know-how and to imbibe discipline, culture and spiritually, and to mould them in to technological giants, dedicated research scientists and intellectual leaders of the country who can spread the beams of light and happiness among the poor and the underprivileged.

ABOUT DEPARTMENT

- Established in: 2002
- Course offered : B.Tech in Computer Science and Engineering

  M.Tech in Computer Science and Engineering

  M.Tech in Cyber Security
- Approved by AICTE New Delhi and Accredited by NAAC
- Affiliated to the University of Dr. A P J Abdul Kalam Technological University.

DEPARTMENT VISION

Producing Highly Competent, Innovative and Ethical Computer Science and Engineering Professionals to facilitate continuous technological advancement.

DEPARTMENT MISSION

1. To Impart Quality Education by creative Teaching Learning Process
2. To Promote cutting-edge Research and Development Process to solve real world problems with emerging technologies.
3. To Inculcate Entrepreneurship Skills among Students.
4. To cultivate Moral and Ethical Values in their Profession.

PROGRAMME EDUCATIONAL OBJECTIVES

PEO1: Graduates will be able to Work and Contribute in the domains of Computer Science and Engineering through lifelong learning.

PEO2: Graduates will be able to Analyse, design and development of novel Software Packages, Web Services, System Tools and Components as per needs and specifications.

PEO3: Graduates will be able to demonstrate their ability to adapt to a rapidly changing environment by learning and applying new technologies.

PEO4: Graduates will be able to adopt ethical attitudes, exhibit effective communication skills, Teamworkand leadership qualities.

PROGRAM OUTCOMES (POS)

Engineering Graduates will be able to:

1. **Engineering knowledge**: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
2. **Problem analysis**: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
3. **Design/development of solutions**: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
4. **Conduct investigations of complex problems**: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
5. **Modern tool usage**: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
6. **The engineer and society**: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
7. **Environment and sustainability**: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
8. **Ethics**: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
9. **Individual and team work**: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
10. **Communication**: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
11. **Project management and finance**: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one’s own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
12. **Life-long learning**: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

PROGRAM SPECIFIC OUTCOMES (PSO)

PSO1: Ability to Formulate and Simulate Innovative Ideas to provide software solutions for Real-time Problems and to investigate for its future scope.

PSO2: Ability to learn and apply various methodologies for facilitating development of high quality System Software Tools and Efficient Web Design Models with a focus on performance optimization.

PSO3: Ability to inculcate the Knowledge for developing Codes and integrating hardware/software products in the domains of Big Data Analytics, Web Applications and Mobile Apps to create innovative career path and for the socially relevant issues.

COURSE OUTCOMES

CO1	Describe fundamentals of C programming such as variables, methods, conditional and iterative execution.
CO2	Analyze and solve programming problems using a procedural and algorithmic approach with functional decomposition
CO3	Design programs that demonstrate effective use of advanced C features including pointers and memory management
CO4	Develop and execute computerized solution for various problems in functions using appropriate C language constructs
CO5	Identify sorting and searching techniques to solve application programs
CO6	Identify and Implement file operations for given application

MAPPING OF COURSE OUTCOMES WITH PROGRAM OUTCOMES

	PO 1	**PO 2**	**PO 3**	**PO 4**	**PO 5**	**PO 6**	**PO 7**	**PO 8**	**PO 9**	**PO 10**	**PO 11**	**PO 12**
CO1	**3**	**3**									**2**	**2**
CO2	**3**	**3**	**2**			**2**			**2**		**2**	**2**
CO3	**2**		**2**	**3**	**3**				**3**			**3**
CO4	**3**	**2**	**3**	**3**	**3**	**2**			**3**		**2**	**3**
CO5	**3**	**2**	**3**	**2**	**3**				**2**		**2**	**2**
CO6	**3**		**3**	**3**	**3**	**2**			**2**			**2**

Note: H-Highly correlated=3, M-Medium correlated=2, L-Less correlated=1

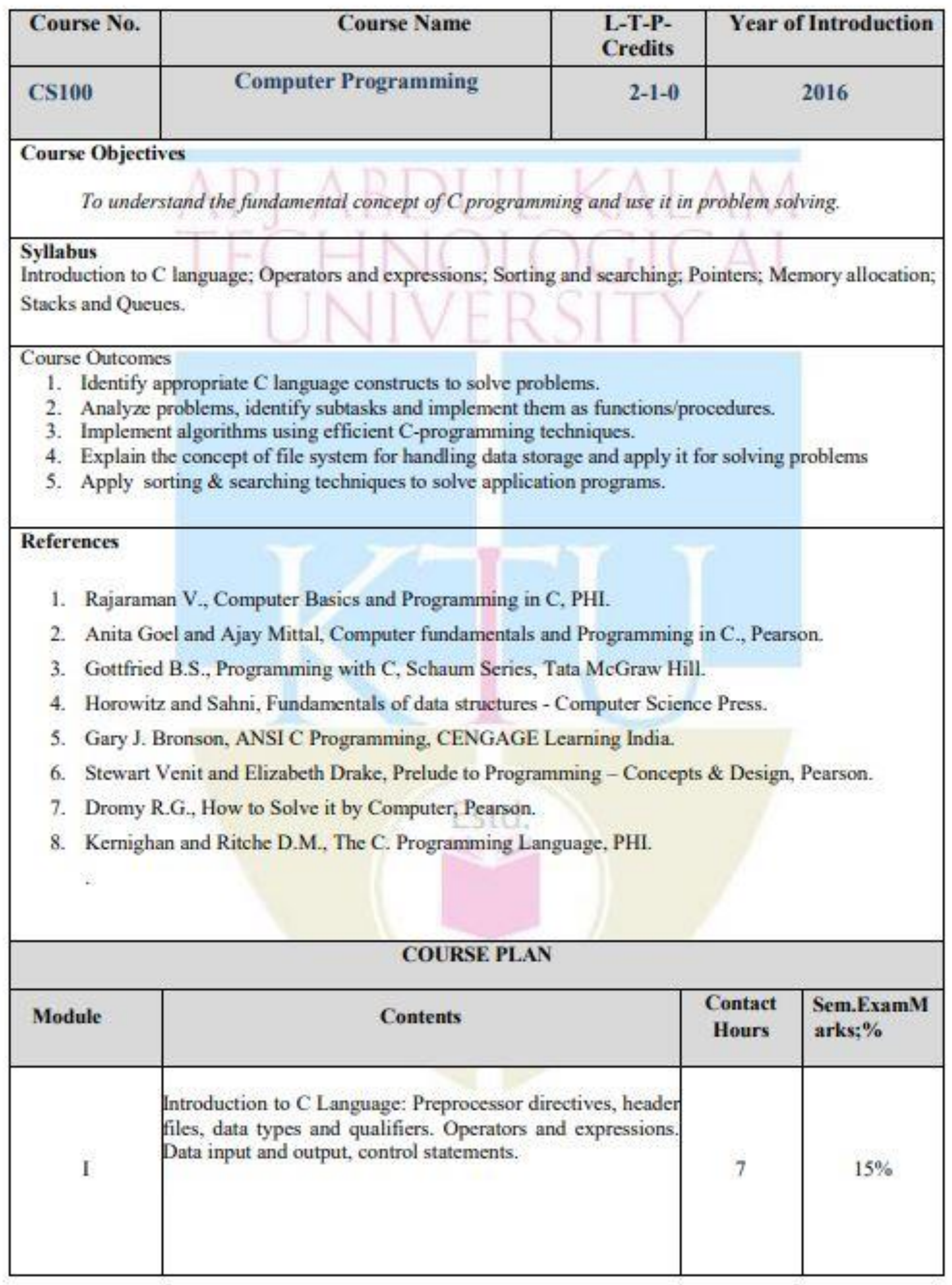
MAPPING OF COURSE OUTCOMES WITH PROGRAM SPECIFIC OUTCOMES

	PSO 1	PSO 2	PSO 3
CO1	3	2	
CO2	3	3	
CO3	3	3	2
CO4	3	3	
CO5	3	3	
CO6	3	3	

Note: H-Highly correlated=3, M-Medium correlated=2, L-Less correlated=1

•

SYLLABUS



Course No.	Course Name	L-T-P-Credits	Year of Introduction
CS100	Computer Programming	2-1-0	2016

Course Objectives

To understand the fundamental concept of C programming and use it in problem solving.

Syllabus

Introduction to C language; Operators and expressions; Sorting and searching; Pointers; Memory allocation; Stacks and Queues.

Course Outcomes

1. Identify appropriate C language constructs to solve problems.
2. Analyze problems, identify subtasks and implement them as functions/procedures.
3. Implement algorithms using efficient C-programming techniques.
4. Explain the concept of file system for handling data storage and apply it for solving problems
5. Apply sorting & searching techniques to solve application programs.

References

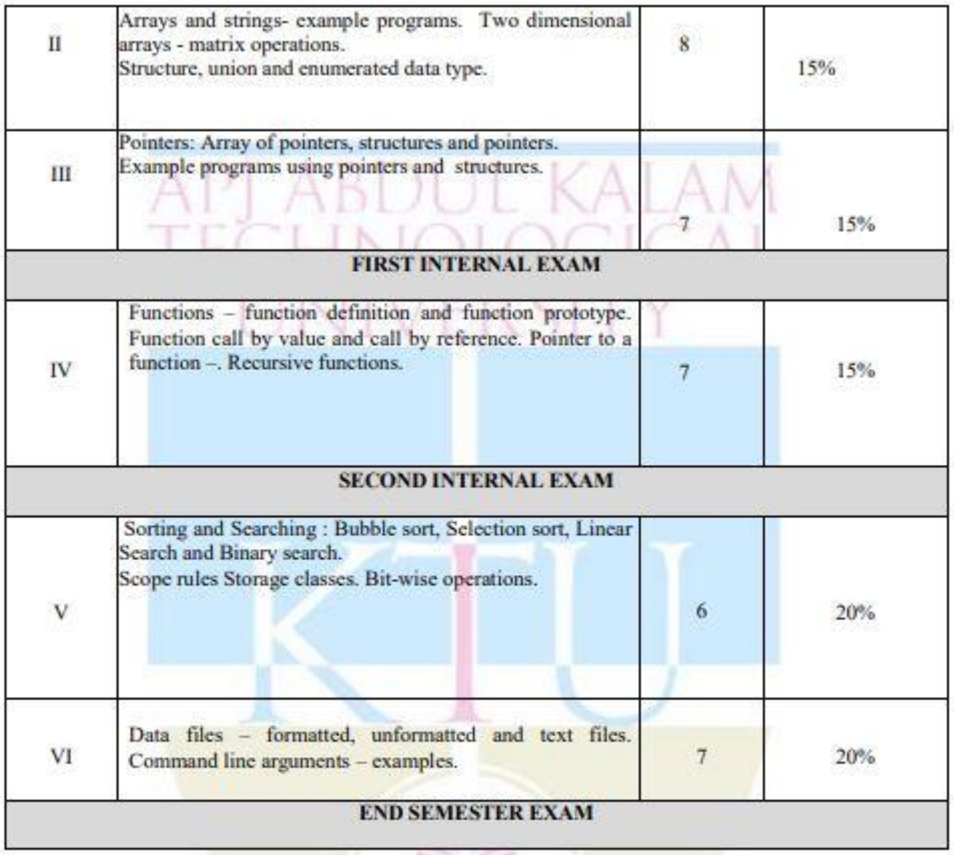
1. Rajaraman V., Computer Basics and Programming in C, PHI.
2. Anita Goel and Ajay Mittal, Computer fundamentals and Programming in C., Pearson.
3. Gottfried B.S., Programming with C, Schaum Series, Tata McGraw Hill.
4. Horowitz and Sahni, Fundamentals of data structures - Computer Science Press.
5. Gary J. Bronson, ANSI C Programming, CENGAGE Learning India.
6. Stewart Venit and Elizabeth Drake, Prelude to Programming – Concepts & Design, Pearson.
7. Dromy R.G., How to Solve it by Computer, Pearson.
8. Kernighan and Ritche D.M., The C. Programming Language, PHI.

.

COURSE PLAN

Module	Contents	Contact Hours	Sem.ExamMarks;%
I	Introduction to C Language: Preprocessor directives, header files, data types and qualifiers. Operators and expressions. Data input and output, control statements.	7	15%

•



II	Arrays and strings- example programs. Two dimensional arrays - matrix operations. Structure, union and enumerated data type.	8	15%
III	Pointers: Array of pointers, structures and pointers. Example programs using pointers and structures.	7	15%
FIRST INTERNAL EXAM			
IV	Functions – function definition and function prototype. Function call by value and call by reference. Pointer to a function –. Recursive functions.	7	15%
SECOND INTERNAL EXAM			
V	Sorting and Searching : Bubble sort, Selection sort, Linear Search and Binary search. Scope rules Storage classes. Bit-wise operations.	6	20%
VI	Data files – formatted, unformatted and text files. Command line arguments – examples.	7	20%
END SEMESTER EXAM			

•

QUESTION BANK

MODULE I				
Q:NO:	**QUESTIONS**	**CO**	**KL**	**PAGE NO:**
1	Differentiate between Keywords and Identifiers in C with examples.	CO1	K3	25
2	Describe about the fundamental data-types in C.	CO1	K2	34
3	Write a C program to print the Fibonacci series.	CO1	K3	61
4	Explain with example, how break and continue constructs are useful in C programming.	CO1	K2	66
5	Explain the working of loop control statements in C with examples.	CO1	K5	58
6	Write a C program to find the largest of three numbers.	CO1	K3	44
7	Point out the different pre-processor directives with its uses and example.	CO1	K4	20

•

8	Write a C program to swap two numbers.	CO1	K3	33
9	Write a C program to find the largest among three given numbers, by applying conditional operator.	CO1	K3	66
10	Write a C program to print the factors of a given number.	CO1	K5	59
11	Write a C program for menu driven calculator.	CO1	K5	53
12	Explain the syntax of *switch* construct in C.	CO1	K2	53
MODULE II				
1	Write a C program to concatenate two strings without using built in function.	CO2	K3	78
2	Write a C program to input the array with 10 characters and print the array in reverse order.	CO2	K6	69
3	Write a C program to input names in an array and sort the names in the increasing ASCII value.	CO2	K6	72

•

4	Compare and Contrast C data structures Structure and Union with example.	CO2	K4	82,94
5	Write a C program to find the count of the inputted character in a given string.	CO2	K3	75
6	Write a C program to sort names in an array in lexicographical order.	CO2	K3	69
7	Write a C program to find the number of vowels, consonants, digits and white space in a string.	CO2	K3	75
8	Write a C program for matrix addition.	CO2	K6	73
9	Distinguish between an array and a structure with example.	CO2	K4	69,82
10	Write a C program to copy a string without built in function.	CO2	K3	75
11	Differentiate between Structure and Union with example.	CO2	K4	82,94
12	Write a C program to find the frequency of character in a given string.	CO2	K3	75
13	Write a C program to find the largest and smallest element in an integer array.	CO2	K3	69

•

14	Write a C program to print an array in reverse order.	CO2	K3	69
MODULE III				
1	Explain the concept of pointers with declaration and initialization.	CO3	K2	103
2	Write a program to sort elements of an array using pointers.	CO3	K6	111
3	Write a program to create an employee structure and access its member variables using pointers.	CO3	K6	122
4	Write a program to swap two numbers using pointers.	CO3	K3	104
5	Write a program to create a structure with name student and member variables name, number and rank and then access the structure members with pointers.	CO3	K6	123
6	Write a program to read a string and print the string in the output terminal using pointers.	CO3	K6	113

•

7	Write a program to read a string and find the number of characters in the string using pointers.	CO3	K6	113
8	Write a program to find the length of a string using pointers.	CO3	K3	114
9	Write a program to concatenate two strings using pointers.	CO3	K3	115
10	Write a program to copy string using pointers.	CO3	K3	113
11	Write a program to print a string using pointers.	CO3	K3	113
12	Write a program to access an array using pointers.	CO3	K3	112
13	Explain the concept of structure arrays in C.	CO3	K5	123
14	Explain the concept of pointer arrays in C.	CO3	K2	103
15	Explain the concept of dynamic memory allocation in C.	CO3	K5	102
16	Point out the advantages and disadvantages of using pointers.	CO3	K4	107

•

17	Point out the benefits of using pointers.	CO3	K4	108
MODULE IV				
1	Describe recursion with an example.	CO4	K2	145
2	Justify the need of functions in C.	CO4	K1	131
3	Write a C program to find factorial of a given number using recursive function.	CO4	K3	145
4	Write a C program to swap numbers by: (a) call by reference and (b) call by value.	CO4	K6	147
5	Write a C program to find the sum of natural numbers using recursion.	CO4	K3	145
6	Write a C program to find the reverse of a number using recursive function.	CO4	K6	145
7	Write a C program to find the sum of digits of a number using recursive function.	CO4	K6	145
8	Explain how the memory allocation is performed dynamically in C.	CO4	K2	102
9	Write a C program using function to check whether a number is Armstrong or not.	CO4	K6	131

•

10	Write a C program to find the reverse of a number using recursive function.	CO4	K6	145
11	Write a C program to find prime number or not using recursion.	CO4	K3	145
12	With suitable example explain different function parameter passing methods in C.	CO4	K2	147
13	Write a C program to check whether a given number is perfect number or not using function.	CO4	K3	145
14	Write a C program to find $^{n}C_{r}$ using function.	CO4	K3	145
15	Write a C program to simulate a menu driven calculator with addition, subtraction, multiplication, division and exponentiation operations. Use a separate function to implement each operation	CO4	K6	131
16	Write a C program to print Fibonacci series using recursion.	CO4	K3	145
MODULE V				
1	Explain the working of selection sort with an example.	CO5	K5	166

•

2	Give the syntax and use of external storage class.	CO5	K2	190
3	Write a C program to search an element in an array using binary search.	CO5	K3	172
4	What is meant by scope of a variable in C?	CO5	K1	180
5	Illustrate the steps of sorting of the following set of numbers using selection sort: 15,10,11,18,12	CO5	K6	166
6	Describe bitwise operations in C.	CO5	K3	196
7	Write a C program to find the second largest element of an unsorted array.	CO5	K3	157
8	Differentiate between linear and binary search techniques in C.	CO5	K4	172
9	Explain register storage class with an example.	CO5	K2	195
10	What are the different storage classes in C? Explain with example.	CO5	K1	188
MODULE VI				

•

1	Explain any Six File opening modes available in C.	CO6	K2	199
2	Write any two file handling functions used to write data into text files.	CO6	K3	202
3	Write a C program to create a file and store information about a person, in terms of his name, age and salary.	CO6	K6	201
4	What is the purpose of fopen( ) and fclose( ) functions in C.	CO6	K1	209
5	What is the purpose of getw( ) and putw( ) function?	CO6	K1	199
6	Discuss the concept of binary file in C.	CO6	K2	204
7	Write a C program to copy the content of a given text file to a new file after replacing every lowercase letters with corresponding uppercase letters.	CO6	K6	207
8	Write a C program to write a set of numbers to a file and separate the odd and even numbers to two separate files.	CO6	K3	199
9	What is an unformatted data file? List the applications of such files.	CO6	K1	204

•

10	Write a C program to copy the contents of a text file to another file. Pass the filename using command line arguments.	CO6	K6	201
11	Discuss about unformatted data files and write on any two library functions associated with this.	CO6	K2	204
12	With suitable example explain any four different File I/O operations in C?	CO6	K5	198

•

APPENDIX 1		
CONTENT BEYOND THE SYLLABUS		
S:NO;	TOPIC	PAGE NO:
1	3D Array	210
2	Enumeration (ENUM) in C	211

•

MODULE 1

Overview of C Language

C is a structured programming language developed by Dennis Ritchie in 1973 at Bell Laboratories. It is one of the most popular computer languages today because of its structure, high-level abstraction, machine independent feature etc. C language was developed to write the UNIX operating system, hence it is strongly associated with UNIX, which is one of the most popular network operating system in use today.

Features of C language

- It is a robust language with rich set of built-in functions and operators that can be used to write any complex program.
- The C compiler combines the capabilities of an assembly language with features of a high-level language.
- Programs Written in C are efficient and fast. This is due to its variety of data type and powerful operators.
- It is many time faster than BASIC.
- C is highly portable this means that programs once written can be run on another machines with little or no modification.
- Another important feature of C program, is its ability to extend itself.
- A C program is basically a collection of functions that are supported by C library. We can also create our own function and add it to C library.

-

- C language is the most widely used language in operating systems and embedded system development today.

Different parts of C program

- Pre-processor
- Header file
- Function
- Variables
- Statements & expressions
- Comments

Pre-processor

`#include` is the first word of any C program. It is also known as a **pre-processor**. The task of a pre-processor is to initialize the environment of the program, i.e to link the program with the header files required.

So, when we say `#include <stdio.h>`, it is to inform the compiler to include the **stdio.h** header file to the program before executing it.

Preprocessor	**Syntax/Description**

•

Macro	**Syntax:** #define This macro defines constant value and can be any of the basic data types.
Header file inclusion	**Syntax:** #include <file_name> The source code of the file "file_name" is included in the main program at the specified place.
Conditional compilation	**Syntax:** #ifdef, #endif, #if, #else, #ifndef Set of commands are included or excluded in source program before compilation with respect to the condition.
Other directives	**Syntax:** #undef, #pragma #undef is used to undefine a defined macro variable. #Pragma is used to call a function before and after main function in a C program.

Header file

A Header file is a collection of built-in(readymade) functions, which we can directly use in our program. Header files contain definitions of the functions which can be incorporated into any C program by using pre-processor `#include` statement with the header file. Standard header files are provided with each compiler, and covers a range of areas like string handling, mathematical functions, data conversion, printing and reading of variables.

•

With time, you will have a clear picture of what header files are, as of now consider as a readymade piece of function which comes packaged with the C language and you can use them without worrying about how they work, all you have to do is include the header file in your program.

To use any of the standard functions, the appropriate header file must be included. This is done at the beginning of the C source file.

For example, to use the `printf()` function in a program, which is used to display anything on the screen, the line `#include <stdio.h>` is required because the header file **stdio.h** contains the `printf()` function. All header files will have an extension `.h`

main() function

`main()` function is a function that must be there in every C program. Everything inside this function in a C program will be executed. In the above example, `int` written before the `main()` function is the **return type** of main() function. we will discuss about it in detail later. The curly braces `{ }` just after the **main()** function encloses the **body** of **main()** function.

.

Comments

We can add comments in our program to describe what we are doing in the program. These comments are ignored by the compiler and are not executed.

To add a single line comment, start it by adding two forward slashses `//` followed by the comment.

To add multiline comment, enclode it between `/*  ....  */`, just like in the program above.

Return statement - return 0;

A return statement is just meant to define the end of any C program.

All the C programs can be written and edited in normal text editors like Notepad or Notepad++ and must be saved with a file name with extension as `.c`

If you do not add the extension `.c` then the compiler will not recognize it as a C language program file.

C Language Basic Syntax Rules

C language syntax specify rules for sequence of characters to be written in C language. In simple language it states how to form statements in a C language

•

program - How should the line of code start, how it should end, where to use double quotes, where to use curly brackets etc.

The rule specify how the character sequence will be grouped together, to form **tokens**. A smallest individual unit in C program is known as **C Token**. Tokens are either keywords, identifiers, constants, variables or any symbol which has some meaning in C language. A C program can also be called as a collection of various tokens.

If we take any one statement:

printf("Hello,World");

Then the tokens in this statement are→ printf, (, "Hello,World",) and ;.

So C tokens are basically the building blocks of a C program.

Semicolon ;

Semicolon ; is used to mark the end of a statement and beginning of another statement. Absence of semicolon at the end of any statement, will mislead the compiler to think that this statement is not yet finished and it will add the next consecutive statement after it, which may lead to compilation(syntax) error.

•

Comments

Comments are plain simple text in a C program that are not compiled by the compiler. We write comments for better understanding of the program. Though writing comments is not compulsory, but it is recommended to make your program more descriptive. It make the code more readable.

There are two ways in which we can write comments.

1. Using // This is used to write a single line comment.
2. Using /* */: The statements enclosed within /* and */ , are used to write multi-line comments.

Some basic syntax rule for C program

- C is a case sensitive language so all C instructions must be written in lower case letter.
- All C statement must end with a semicolon.
- Whitespace is used in C to describe blanks and tabs.
- Whitespace is required between keywords and identifiers. We will learn about keywords and identifiers in the next tutorial.

What are Keywords in C?

Keywords are preserved words that have special meaning in C language. The meaning of C language keywords has already been described to the C compiler. These meaning cannot be changed. Thus, keywords cannot be used as variable

•

names because that would try to change the existing meaning of the keyword, which is not allowed. There are total 32 keywords in C language.

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

•

<u>What are Identifiers?</u>

In C language identifiers are the names given to variables, constants, functions and user-define data. These identifier are defined against a set of rules.

Rules for an Identifier

1. An Identifier can only have alphanumeric characters (a-z , A-Z , 0-9) and underscore(_).
2. The first character of an identifier can only contain alphabet (a-z , A-Z) or underscore (_).
3. Identifiers are also case sensitive in C. For example **name** and **Name** are two different identifiers in C.
4. Keywords are not allowed to be used as Identifiers.
5. No special characters, such as semicolon, period, whitespaces, slash or comma are permitted to be used in or as Identifier.

•

OPERATORS IN C LANGUAGE

C language supports a rich set of built-in operators. An operator is a symbol that tells the compiler to perform a certain mathematical or logical manipulation. Operators are used in programs to manipulate data and variables.

C operators can be classified into following types:

- Arithmetic operators
- Relational operators
- Logical operators
- Bitwise operators
- Assignment operators
- Conditional operators
- Special operators

Arithmetic operators

C supports all the basic arithmetic operators. The following table shows all the basic arithmetic operators.

Operator	Description
+	adds two operands

•

-	subtract second operands from first
*	multiply two operand
/	divide numerator by denominator
%	remainder of division
++	Increment operator - increases integer value by one
--	Decrement operator - decreases integer value by one

Relational operators

The following table shows all relation operators supported by C.

Operator	**Description**
==	Check if two operand are equal
!=	Check if two operand are not equal.

•

>	Check if operand on the left is greater than operand on the right
<	Check operand on the left is smaller than right operand
>=	check left operand is greater than or equal to right operand
<=	Check if operand on left is smaller than or equal to right operand

Logical operators

C language supports following 3 logical operators. Suppose `a = 1` and `b = 0`,

Operator	Description	Example
&&	Logical AND	(a && b) is false
||	Logical OR	(a || b) is true
!	Logical NOT	(!a) is false

•

Bitwise operators

Bitwise operators perform manipulations of data at **bit level**. These operators also perform **shifting of bits** from right to left. Bitwise operators are not applied to `float` or `double`

Operator	Description
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
<<	left shift
>>	right shift

Now lets see truth table for bitwise `&`, `|` and `^`

a	b	a & b	a | b	a ^ b
0	0	0	0	0
0	1	0	1	1

•

1	0	0	1	1
1	1	1	1	0

The bitwise **shift** operator, shifts the bit value. The left operand specifies the value to be shifted and the right operand specifies the number of positions that the bits in the value have to be shifted. Both operands have the same precedence.

Assignment Operators

Assignment operators supported by C language are as follows.

Operator	**Description**	**Example**
=	assigns values from right side operands to left side operand	a=b
+=	adds right operand to the left operand and assign the result to left	a+=b is same as a=a+b
-=	subtracts right operand from the left operand and assign the result to left operand	a-=b is same as a=a-b
*=	mutiply left operand with the right operand and assign the result to left operand	a*=b is same as a=a*b

•

/=	divides left operand with the right operand and assign the result to left operand	a/=b is same as a=a/b
%=	calculate modulus using two operands and assign the result to left operand	a%=b is same as a=a%b

Conditional operator

The conditional operators in C language are known by two more names

1. **Ternary Operator**
2. **? : Operator**

It is actually the `if` condition that we use in C language decision making, but using conditional operator, we turn the `if` condition statement into a short and simple operator.

The syntax of a conditional operator is :

expression 1 ? expression 2: expression 3

Explanation:

- The question mark **"?"** in the syntax represents the **if** part.
- The first expression (expression 1) generally returns either true or false, based on which it is decided whether (expression 2) will be executed or (expression 3)

-

- If (expression 1) returns true then the expression on the left side of **" : "** i.e (expression 2) is executed.
- If (expression 1) returns false then the expression on the right side of **" : "** i.e (expression 3) is executed.

Special operator

Operator	Description	Example
sizeof	Returns the size of an variable	**sizeof(x)** return size of the variable **x**
&	Returns the address of an variable	**&x ;** return address of the variable **x**
*	Pointer to a variable	***x ;** will be pointer to a variable **x**

DATA TYPES IN C LANGUAGE

Data types specify how we enter data into our programs and what type of data we enter. C language has some predefined set of data types to handle various kinds of data that we can use in our program. These datatypes have different storage capacities.

•

C language supports 2 different type of data types:

1. **Primary data types**:

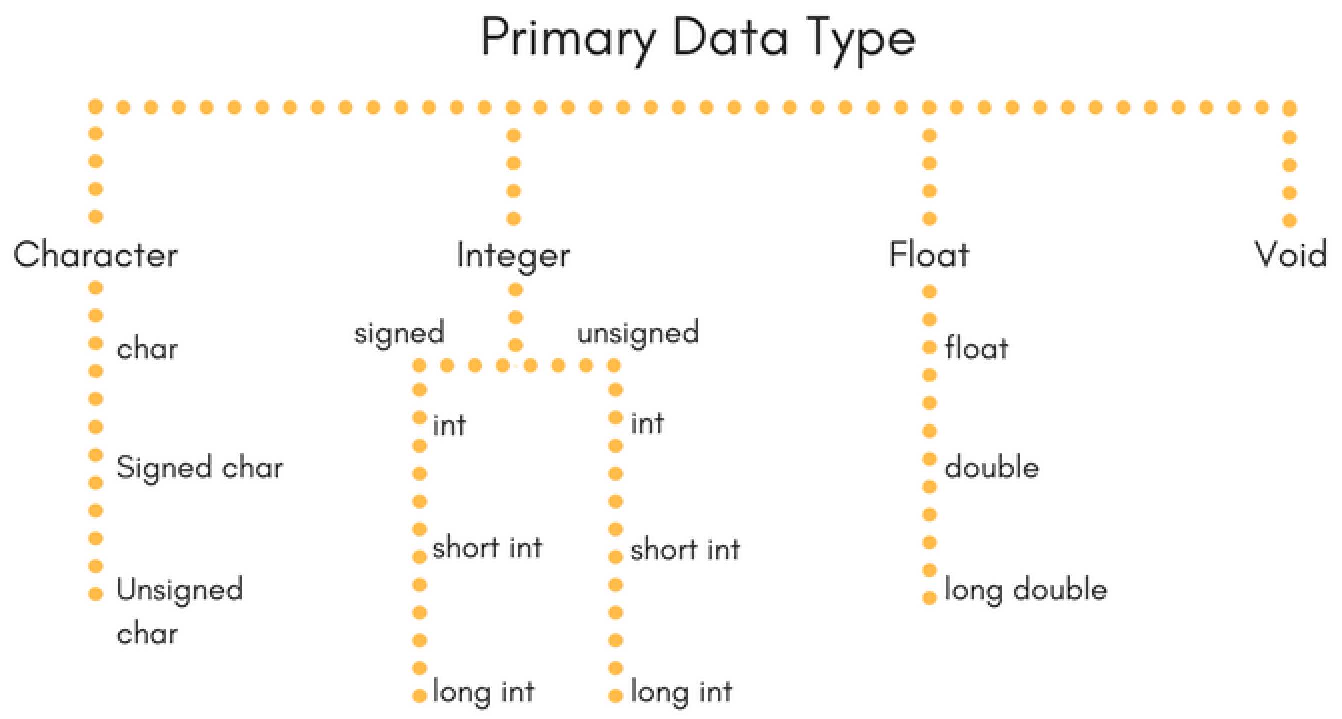
   These are fundamental data types in C namely integer(`int`), floating point(`float`), character(`char`) and `void`.

2. **Derived data types**

   Derived data types are nothing but primary datatypes but a little twisted or grouped together like **array**, **stucture**, **union** and **pointer**. These are discussed in details later.

Data type determines the type of data a variable will hold. If a variable `x` is declared as `int`. it means x can hold only integer values. Every variable which is used in the program must be declared as what data-type it is.

•



Integer type

Integers are used to store whole numbers.

Size and range of Integer type on 16-bit machine:

Type	Size(bytes)	Range

•

int or signed int	2	-32,768 to 32767
unsigned int	2	0 to 65535
short int or signed short int	1	-128 to 127
unsigned short int	1	0 to 255
long int or signed long int	4	-2,147,483,648 to 2,147,483,647
unsigned long int	4	0 to 4,294,967,295

Floating point type

Floating types are used to store real numbers.

Size and range of Integer type on 16-bit machine

Type	Size(bytes)	Range
Float	4	3.4E-38 to 3.4E+38
double	8	1.7E-308 to 1.7E+308

•

long double	10	3.4E-4932 to 1.1E+4932

Character type

Character types are used to store characters value.

Size and range of Integer type on 16-bit machine

Type	Size(bytes)	Range
char or signed char	1	-128 to 127
unsigned char	1	0 to 255

void type

`void` type means no value. This is usually used to specify the type of functions which returns nothing. We will get acquainted to this datatype as we start learning more advanced topics in C language, like functions, pointers etc.

VARIABLES IN C LANGUAGE

When we want to store any information (data) on our computer/laptop, we store it in the computer's memory space. Instead of remembering the complex address of

•

that memory space where we have stored our data, our operating system provides us with an option to create folders, name them, so that it becomes easier for us to find it and access it.

Similarly, in C language, when we want to use some data value in our program, we can store it in a memory space and name the memory space so that it becomes easier to access it.

The naming of an address is known as **variable**. Variable is the name of memory location. Unlike constant, variables are changeable, we can change value of a variable during execution of a program. A programmer can choose a meaningful variable name. Example : average, height, age, total etc.

Datatype of Variable

A **variable** in C language must be given a type, which defines what type of data the variable will hold.

It can be:

- char: Can hold/store a character in it.
- int: Used to hold an integer.
- float: Used to hold a float value.
- double: Used to hold a double value.
- void

•

Rules to name a Variable

1. Variable name must not start with a digit.
2. Variable name can consist of alphabets, digits and special symbols like underscore _.
3. Blank or spaces are not allowed in variable name.
4. Keywords are not allowed as variable name.
5. Upper and lower case names are treated as different, as C is case-sensitive, so it is suggested to keep the variable names in lower case.

Declaring, Defining and Initializing a variable

Declaration of variables must be done before they are used in the program. Declaration does the following things.

1. It tells the compiler what the variable name is.
2. It specifies what type of data the variable will hold.
3. Until the variable is defined the compiler doesn't have to worry about allocating memory space to the variable.
4. Declaration is more like informing the compiler that there exist a variable with following datatype which is used in the program.
5. A variable is declared using the `extern` keyword, outside the `main()` function.

.

C INPUT AND OUTPUT

Input means to provide the program with some data to be used in the program and **Output** means to display data on screen or write the data to a printer or a file.

C programming language provides many built-in functions to read any given input and to display data on screen when there is a need to output the result.

scanf() and printf() functions

The standard input-output header file, named `stdio.h` contains the definition of the functions `printf()` and `scanf()`, which are used to display output on screen and to take input from user respectively.

`%d` inside the `scanf()` or `printf()` functions. It is known as **format string** and this informs the `scanf()` function, what type of input to expect and in `printf()` it is used to give a heads up to the compiler, what type of output to expect.

Format String	**Meaning**
`%d`	Scan or print an integer as signed decimal number
`%f`	Scan or print a floating point number
`%c`	To scan or print a character

•

`%s`	To scan or print a character string. The scanning ends at whitespace.

We can also **limit the number of digits or characters** that can be input or output, by adding a number with the format string specifier, like `"%1d"` or `"%3s"`, the first one means a single numeric digit and the second one means 3 characters, hence if you try to input `42`, while `scanf()` has `"%1d"`, it will take only `4` as input. Same is the case for output.

getchar() & putchar() functions

The `getchar()` function reads a character from the terminal and returns it as an integer. This function reads only single character at a time. You can use this method in a loop in case you want to read more than one character. The `putchar()` function displays the character passed to it on the screen and returns the same character. This function too displays only a single character at a time. In case you want to display more than one characters, use `putchar()` method in a loop.

gets() & puts() functions

The `gets()` function reads a line from **stdin**(standard input) into the buffer pointed to by `str` pointer, until either a terminating newline or EOF (end of file) occurs. The `puts()` function writes the string `str` and a trailing newline to **stdout**.

.

Difference between scanf() and gets()

The main difference between these two functions is that `scanf()` stops reading characters when it encounters a space, but `gets()` reads space as character too.

If you enter name as **Study Tonight** using `scanf()` it will only read and store **Study** and will leave the part after space. But `gets()` function will read it completely.

CONTROL STATEMENTS

- Decision making statements
- Selection statements
- Iteration or Looping statements
- Jumping statements

Decision making with if statement

The `if` statement may be implemented in different forms depending on the complexity of conditions to be tested. The different forms are,

1. Simple if statement
2. if....else statement
3. Nested if....else statement

•

4. Using else if statement

Simple if statement

The general form of a simple `if` statement is,

```
if(expression)

{

    statement inside;

}

    statement outside;
```

If the *expression* returns true, then the **statement-inside** will be executed, otherwise **statement-inside** is skipped and only the **statement-outside** is executed.

Example:

```
#include <stdio.h>

void main( )

{

    int x, y;

    x = 15;

    y = 13;

    if (x > y )

    {
```

•

```
        printf("x is greater than y");
    }
}
```

```
x is greater than y
```

if...else statement

The general form of a simple `if...else` statement is,

```
if(expression)
{
    statement block1;
}
else
{
    statement block2;
}
```

If the *expression* is true, the **statement-block1** is executed, else **statement-block1** is skipped and **statement-block2** is executed.

Example:

•

```
#include <stdio.h>


void main( )

{

    int x, y;

    x = 15;

    y = 18;

    if (x > y )

    {

        printf("x is greater than y");

    }

    else

    {

        printf("y is greater than x");

    }

}
```

```
y is greater than x
```

•

Nested if....else statement

The general form of a nested `if...else` statement is,

```
if( expression )
{
   if( expression1 )
   {
      statement block1;
   }
   else
   {
      statement block2;
   }
}
else
{
   statement block3;
}
```

if *expression* is false then **statement-block3** will be executed, otherwise the execution continues and enters inside the first `if` to perform the check for the

•

next `if` block, where if *expression 1* is true the **statement-block1** is executed otherwise **statement-block2** is executed.

Example:

```
#include <stdio.h>


void main( )

{

    int a, b, c;

    printf("Enter 3 numbers...");

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

    if(a > b)

    {

        if(a > c)

        {

            printf("a is the greatest");

        }

        else

        {
```

•

```
            printf("c is the greatest");
        }
    }
    else
    {
        if(b > c)
        {
            printf("b is the greatest");
        }
        else
        {
            printf("c is the greatest");
        }
    }
}
```

else if ladder

The general form of else-if ladder is,

•

```
if(expression1)
{
    statement block1;
}
else if(expression2)
{
    statement block2;
}
else if(expression3 )
{
    statement block3;
}
else
    default statement;
```

The expression is tested from the top(of the ladder) downwards. As soon as a **true** condition is found, the statement associated with it is executed.

Example :

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

•

```
void main( )
{
    int a;
    printf("Enter a number...");
    scanf("%d", &a);
    if(a%5 == 0 && a%8 == 0)
    {
        printf("Divisible by both 5 and 8");
    }
    else if(a%8 == 0)
    {
        printf("Divisible by 8");
    }
    else if(a%5 == 0)
    {
        printf("Divisible by 5");
    }
    else
```

•

```
    {
        printf("Divisible by none");
    }
}
```

Points to Remember

1. In if statement, a single statement can be included without enclosing it into curly braces { ... }

```
int a = 5;
```

```
if(a > 4)
    printf("success");
```

2. No curly braces are required in the above case, but if we have more than one statement inside `if`condition, then we must enclose them inside curly braces.
3. == must be used for comparison in the expression of if condition, if you use = the expression will always return **true**, because it performs assignment not comparison.
4. Other than **0(zero)**, all other values are considered as **true**.

```
if(27)
```

•

```
   printf("hello");
```

In above example, **hello** will be printed.

Switch statement in C (Selection statement)

When you want to solve multiple option type problems, for example: Menu like program, where one value is associated with each option and you need to choose only one at a time, then, `switch` statement is used.

Switch statement is a control statement that allows us to choose only one choice among the many given choices. The expression in `switch` evaluates to return an integral value, which is then compared to the values present in different cases. It executes that block of code which matches the case value. If there is no match, then **default** block is executed(if present). The general form of `switch` statement is,

```
switch(expression)
{
   case value-1:
           block-1;
           break;
   case value-2:
           block-2;
```

•

```
          break;
   case value-3:
          block-3;
          break;
   case value-4:
          block-4;
             break;
   default:
             default-block;
          break;
}
```

Rules for using switch statement

1. The expression (after switch keyword) must yield an **integer** value i.e the expression should be an integer or a variable or an expression that evaluates to an integer.
2. The case **label** values must be unique.
3. The case label must end with a colon(:)
4. The next line, after the **case** statement, can be any valid C statement.

•

Points to Remember

1. We don't use those expressions to evaluate switch case, which may return floating point values or strings or characters.
2. break statements are used to **exit** the switch block. It isn't necessary to use break after each block, but if you do not use it, then all the consecutive blocks of code will get executed after the matching block.

```
int i = 1;
switch(i)
{
   case 1:
      printf("A");        // No break
   case 2:
      printf("B");        // No break
   case 3:
      printf("C");
      break;
}
```

•

```
A B C
```

The output was supposed to be only **A** because only the first case matches, but as there is no `break` statement after that block, the next blocks are executed too, until it a `break` statement in encountered or the execution reaches the end of the `switch` block.

3. **default** case is executed when none of the mentioned case matches the switch expression. The default case can be placed anywhere in the switch case. Even if we don't include the default case, switch statement works.
4. Nesting of switch statements are allowed, which means you can have switch statements inside another switch block. However, nested switch statements should be avoided as it makes the program more complex and less readable.

Example of switch statement

```
#include<stdio.h>

void main( )

{

   int a, b, c, choice;

   while(choice != 3)
```

•

```
{
    /* Printing the available options */
    printf("\n 1. Press 1 for addition");
    printf("\n 2. Press 2 for subtraction");
    printf("\n Enter your choice");
    /* Taking users input */
    scanf("%d", &choice);

    switch(choice)
    {
        case 1:
            printf("Enter 2 numbers");
            scanf("%d%d", &a, &b);
            c = a + b;
            printf("%d", c);
            break;
        case 2:
            printf("Enter 2 numbers");
            scanf("%d%d", &a, &b);
```

•

```
            c = a - b;

            printf("%d", c);

            break;

         default:

            printf("you have passed a wrong key");

            printf("\n press any key to continue");

      }

   }

}
```

Difference between switch and if

- if statements can evaluate float conditions. switch statements cannot evaluate floatconditions.
- if statement can evaluate relational operators. switch statement cannot evaluate relational operators i.e they are not allowed in switch statement.

Looping Statements

•

Types of Loop

There are 3 types of Loop in C language, namely:

1. while loop
2. for loop
3. do while loop

while loop

while loop can be addressed as an **entry control** loop. It is completed in 3 steps.

- Variable initialization.(e.g int x = 0;)
- condition(e.g while(x <= 10))
- Variable increment or decrement (x++ or x-- or x = x + 2)

Syntax :

```
variable initialization;

while(condition)

{

    statements;

    variable increment or decrement;

}
```

•

Example: Program to print first 10 natural numbers

```
#include<stdio.h>


void main( )

{

    int x;

    x = 1;

    while(x <= 10)

    {

        printf("%d\t", x);

        /* below statement means, do x = x+1, increment x by 1*/

        x++;

    }

}
```

```
1 2 3 4 5 6 7 8 9 10
```

•

for loop

`for` loop is used to execute a set of statements repeatedly until a particular condition is satisfied. We can say it is an **open ended loop.**. General format is,

```
for(initialization; condition; increment/decrement)

{

    statement-block;

}
```

In `for` loop we have exactly two semicolons, one after initialization and second after the condition. In this loop we can have more than one initialization or increment/decrement, separated using comma operator. But it can have only one **condition**.

The `for` loop is executed as follows:

1. It first evaluates the initialization code.
2. Then it checks the condition expression.
3. If it is **true**, it executes the for-loop body.
4. Then it evaluate the increment/decrement condition and again follows from step 2.
5. When the condition expression becomes **false**, it exits the loop.

•

Example: Program to print first 10 natural numbers

```
#include<stdio.h>


void main( )
{
    int x;
    for(x = 1; x <= 10; x++)
    {
        printf("%d\t", x);
    }
}
```

```
1 2 3 4 5 6 7 8 9 10
```

Nested `for` loop

We can also have nested `for` loops, i.e one `for` loop inside another `for` loop. Basic syntax is,

```
for(initialization; condition; increment/decrement)
{
```

•

```
    for(initialization; condition; increment/decrement)
    {
        statement ;
    }
}
```

Example: Program to print half Pyramid of numbers

```
#include<stdio.h>


void main( )
{
    int i, j;
    /* first for loop */
    for(i = 1; i < 5; i++)
    {
        printf("\n");
        /* second for loop inside the first */
        for(j = i; j > 0; j--)
```

•

```
        {
            printf("%d", j);
        }
    }
}
```

```
1
21
321
4321
54321
```

---

do while loop

In some situations it is necessary to execute body of the loop before testing the condition. Such situations can be handled with the help of `do-while` loop. `do` statement evaluates the body of the loop first and at the end, the condition is checked using `while` statement. It means that the body of the loop will be executed at least once, even though the starting condition inside `while` is initialized to be **false**. General syntax is,

•

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

Example: Program to print first 10 multiples of 5.

```
#include<stdio.h>

void main()
{
    int a, i;
    a = 5;
    i = 1;
    do
    {
        printf("%d\t", a*i);
        i++;
```

•

```
    }
    while(i <= 10);
}
```

```
5 10 15 20 25 30 35 40 45 50
```

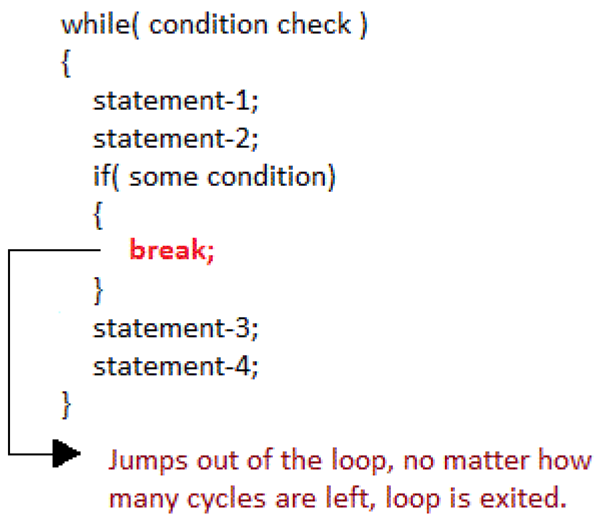
Jumping Statements

Sometimes, while executing a loop, it becomes necessary to skip a part of the loop or to leave the loop as soon as certain condition becomes **true**. This is known as jumping out of loop.

1) break statement

When `break` statement is encountered inside a loop, the loop is immediately exited and the program continues with the statement immediately following the loop.

•

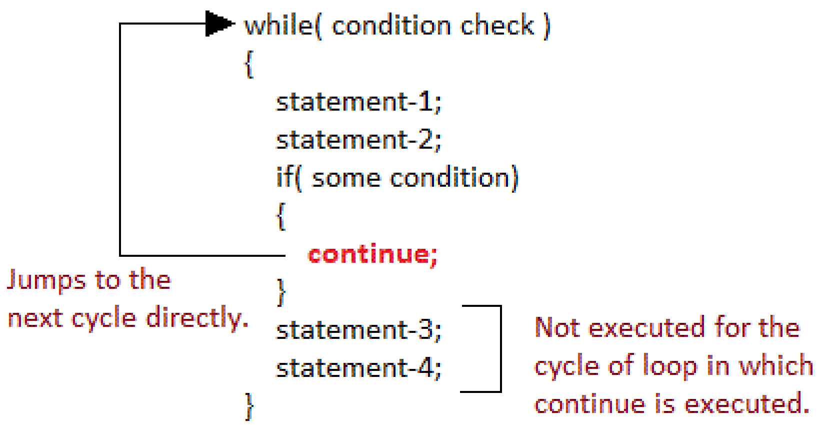


```
while( condition check )
{
   statement-1;
   statement-2;
   if( some condition)
   {
      break;
   }
   statement-3;
   statement-4;
}
```

Jumps out of the loop, no matter how many cycles are left, loop is exited.

2) continue statement

It causes the control to go directly to the test-condition and then continue the loop process. On encountering `continue`, cursor leave the current cycle of loop, and starts with the next cycle.



```
while( condition check )
{
   statement-1;
   statement-2;
   if( some condition)
   {
      continue;
   }
   statement-3;
   statement-4;
}
```

Jumps to the next cycle directly.
Not executed for the cycle of loop in which continue is executed.

•

MODULE 2

Arrays in C

In C language, `arrays` are reffered to as structured data types. An array is defined as finite ordered collection of homogenous data, stored in contiguous memory locations. Arrays are used:

- to store list of Employee or Student names,
- to store marks of students,
- or to store list of numbers or characters etc.

Since arrays provide an easy way to represent data, it is classified amongst the data structures in C.

Array can be used to represent not only simple list of data but also table of data in two or three dimensions.

Declaring an Array

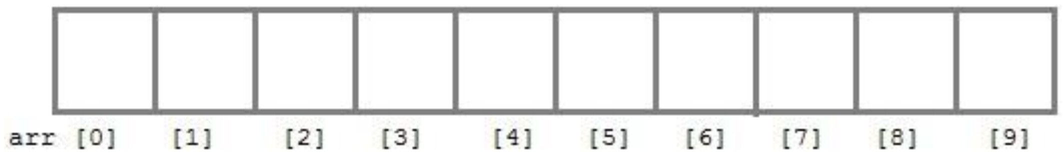
Like any other variable, arrays must be declared before they are used. General form of array declaration is,

•

data-type variable-name[size];

/* Example of array declaration */

int arr[10];



Here int is the data type, arr is the name of the array and 10 is the size of array. It means array arr can only contain 10 elements of int type.

Index of an array starts from 0 to size-1 i.e first element of arr array will be stored at arr[0]address and the last element will occupy arr[9].

Initialization of an Array

After an array is declared it must be initialized. Otherwise, it will contain garbage value(any random value). An array can be initialized at either compile time or at runtime.

•

Compile time Array initialization

Compile time initialization of array elements is same as ordinary variable initialization. The general form of initialization of array is,

```
data-type array-name[size] = { list of values };


/* Here are a few examples */
int marks[4]={ 67, 87, 56, 77 };    // integer array initialization


float area[5]={ 23.4, 6.8, 5.5 };   // float array initialization


int marks[4]={ 67, 87, 56, 77, 59 };    // Compile time error
```

One important thing to remember is that when you will give more initializer(array elements) than the declared array size than the **compiler** will give an error.

```
#include<stdio.h>
void main()
{
    int i;
    int arr[] = {2, 3, 4};      // Compile time array initialization
    for(i = 0 ; i < 3 ; i++)
    {
```

•

```
        printf("%d\t",arr[i]);
    }
}
```

OUTPUT:

```
2 3 4
```

Runtime Array initialization

An array can also be initialized at runtime using scanf() function. This approach is usually used for initializing large arrays, or to initialize arrays with user specified values. Example,

```
#include<stdio.h>

void main()
{
    int arr[4];
    int i, j;
    printf("Enter array element");
    for(i = 0; i < 4; i++)
    {
        scanf("%d", &arr[i]);    //Run time array initialization
```

•

```
    }
    for(j = 0; j < 4; j++)
    {
        printf("%d\n", arr[j]);
    }
}
```

Two dimensional Arrays

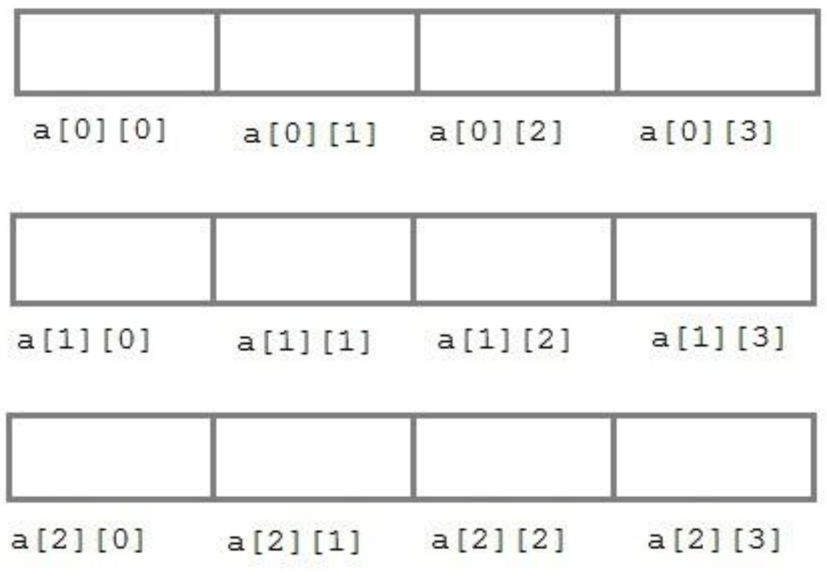
C language supports multidimensional arrays also. The simplest form of a multidimensional array is the two-dimensional array. Both the row's and column's index begins from `0`.

Two-dimensional arrays are declared as follows,

data-type array-name[row-size][column-size]

```
/* Example */
int a[3][4];
```

•



An array can also be declared and initialized together. For example,

```
int arr[][3] = {
    {0,0,0},
    {1,1,1}
};
```

Runtime initialization of a two dimensional Array

```
#include<stdio.h>

void main()
{
```

•

```
    int arr[3][4];
    int i, j, k;
    printf("Enter array element");
    for(i = 0; i < 3;i++)
    {
        for(j = 0; j < 4; j++)
        {
            scanf("%d", &arr[i][j]);
        }
    }
    for(i = 0; i < 3; i++)
    {
        for(j = 0; j < 4; j++)
        {
            printf("%d", arr[i][j]);
        }
    }
}
```

String and Character Array

•

String is a sequence of characters that is treated as a single data item and terminated by null character '\0'. Remember that C language does not support strings as a data type.

A string is actually one-dimensional array of characters in C language. These are often used to create meaningful and readable programs.

For example: The string "hello world" contains 12 characters including '\0' character which is automatically added by the compiler at the end of the string.

Declaring and Initializing a string variables

There are different ways to initialize a character array variable.

```
char name[13] = "StudyTonight";       // valid character array initialization


char name[10] = {'L','e','s','s','o','n','s','\0'};     // valid initialization
```

Remember that when you initialize a character array by listing all of its characters separately then you must supply the '\0' character explicitly.

Some examples of illegal initialization of character array are,

```
char ch[3] = "hell";    // Illegal

char str[4];

str = "hell";   // Illegal
```

•

String Input and Output

Input function scanf() can be used with %s format specifier to read a string input from the terminal. But there is one problem with scanf() function, it terminates its input on the first white space it encounters. Therefore if you try to read an input string "Hello World" using scanf() function, it will only read Hello and terminate after encountering white spaces.

However, C supports a format specification known as the edit set conversion code %[..] that can be used to read a line containing a variety of characters, including white spaces.

```
#include<stdio.h>

#include<string.h>


void main()

{

    char str[20];

    printf("Enter a string");

    scanf("%[^\n]", &str);  //scanning the whole string, including the white spaces

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

•

}

Another method to read character string with white spaces from terminal is by using the gets()function.

char text[20];

gets(text);

printf("%s", text);

String Handling Functions

C language supports a large number of string handling functions that can be used to carry out many of the string manipulations. These functions are packaged in string.h library. Hence, you must include string.h header file in your programs to use these functions.

The following are the most commonly used string handling functions.

Method	Description
strcat()	It is used to concatenate(combine) two strings
strlen()	It is used to show length of a string
strrev()	It is used to show reverse of a string

•

strcpy()	Copies one string into another
strcmp()	It is used to compare two string

(i) strcat() function

strcat("hello", "world");

strcat() function will add the string "world" to "hello" i.e it will ouput helloworld.

(ii) strlen() function

strlen() function will return the length of the string passed to it.

```
int j;

j = strlen("studytonight");

printf("%d",j);
```

OUTPUT:

```
12
```

(iii) strcmp() function

•

strcmp() function will return the ASCII difference between first unmatching character of two strings.

```
int j;
j = strcmp("study", "tonight");
printf("%d",j);
```

OUTPUT:

```
-1
```

(iv) strcpy() function

It copies the second string argument to the first string argument.

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

int main()
{
        char s1[50];
        char s2[50];
         strcpy(s1, "StudyTonight");     //copies "studytonight" to string s1
         strcpy(s2, s1);     //copies string s1 to string s2
          printf("%s\n", s2);
```

•

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

OUTPUT:

```
StudyTonight
```

(v) strrev() function

It is used to reverse the given string expression.

```
#include<stdio.h>
int main()
{
        char s1[50];
         printf("Enter your string: ");
         gets(s1);
         printf("\nYour reverse string is: %s",strrev(s1));
         return(0);
}
```

OUTPUT:

```
Enter your string: studytonight
Your reverse string is: thginotyduts
```

•

Introduction to Structure

Structure is a user-defined datatype in C language which allows us to combine data of different types together. Structure helps to construct a complex data type which is more meaningful.

It is somewhat similar to an Array, but an array holds data of similar type only. But structure on the other hand, can store data of any type, which is practical more useful.

For example: If I have to write a program to store Student information, which will have Student's name, age, branch, permanent address, father's name etc, which included string values, integer values etc, how can I use arrays for this problem, I will require something which can hold data of different types together.

In structure, data is stored in form of records.

Defining a structure

struct keyword is used to define a structure. struct defines a new data type which is a collection of primary and derived datatypes.

Syntax:

struct [structure_tag]

•

```
{
    //member variable 1
    //member variable 2
    //member variable 3
    ...
}[structure_variables];
```

As you can see in the syntax above, we start with the struct keyword, then it's optional to provide your structure a name, we suggest you to give it a name, then inside the curly braces, we have to mention all the member variables, which are nothing but normal C language variables of different types like int, float, array etc.

After the closing curly brace, we can specify one or more structure variables, again this is optional.

Note: The closing curly brace in the structure type declaration must be followed by a semicolon(;).

Example of Structure

```
struct Student
{
    char name[25];
```

•

```
    int age;

    char branch[10];

    // F for female and M for male

    char gender;

};
```

Here struct Student declares a structure to hold the details of a student which consists of 4 data fields, namely name, age, branch and gender. These fields are called structure elements or members.

Each member can have different datatype, like in this case, name is an array of char type and age is of int type etc. Student is the name of the structure and is called as the structure tag.

Declaring Structure Variables

It is possible to declare variables of a structure, either along with structure definition or after the structure is defined. Structure variable declaration is similar to the declaration of any normal variable of any other datatype. Structure variables can be declared in following two ways:

1) Declaring Structure variables separately

•

```
struct Student
{
    char name[25];
    int age;
    char branch[10];
    //F for female and M for male
    char gender;
};

struct Student S1, S2;      //declaring variables of struct Student
```

2) Declaring Structure variables with structure definition

```
struct Student
{
    char name[25];
    int age;
    char branch[10];
```

•

```
    //F for female and M for male
    char gender;
}S1, S2;
```

Here S1 and S2 are variables of structure Student. However this approach is not much recommended.

Accessing Structure Members

Structure members can be accessed and assigned values in a number of ways. Structure members have no meaning individually without the structure. In order to assign a value to any structure member, the member name must be linked with the structure variable using a dot . operator also called period or member access operator.

For example:

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


struct Student
{
    char name[25];
```

```
•

    int age;

    char branch[10];

    //F for female and M for male

    char gender;

};


int main()

{

    struct Student s1;


    /*

        s1 is a variable of Student type and

        age is a member of Student

    */

    s1.age = 18;

    /*

        using string function to add name
```

•

```
    */
    strcpy(s1.name, "Viraaj");
    /*
        displaying the stored values
    */
    printf("Name of Student 1: %s\n", s1.name);
    printf("Age of Student 1: %d\n", s1.age);

    return 0;
}
```

OUTPUT:

```
Name of Student 1: Viraaj
Age of Student 1: 18
```

We can also use scanf() to give values to structure members through terminal.

```
scanf(" %s ", s1.name);
scanf(" %d ", &s1.age);
```

•

Structure Initialization

Like a variable of any other datatype, structure variable can also be initialized at compile time.

```
struct Patient

{

    float height;

    int weight;

    int age;

};


struct Patient p1 = { 180.75 , 73, 23 };    //initialization

or,

struct Patient p1;

p1.height = 180.75;     //initialization of each member separately

p1.weight = 73;

p1.age = 23;
```

Array of Structure

We can also declare an array of structure variables. in which each element of the array will represent a structure variable. Example : struct employee emp[5];

•

The below program defines an array emp of size 5. Each element of the array emp is of type Employee.

```
#include<stdio.h>


struct Employee

{

    char ename[10];

    int sal;

};


struct Employee emp[5];

int i, j;

void ask()

{

    for(i = 0; i < 3; i++)

    {

        printf("\nEnter %dst Employee record:\n", i+1);
```

•

```
        printf("\nEmployee name:\t");
        scanf("%s", emp[i].ename);
        printf("\nEnter Salary:\t");
        scanf("%d", &emp[i].sal);
    }
    printf("\nDisplaying Employee record:\n");
    for(i = 0; i < 3; i++)
    {
        printf("\nEmployee name is %s", emp[i].ename);
        printf("\nSlary is %d", emp[i].sal);
    }
}
void main()
{
    ask();
}
```

•

Nested Structures

Nesting of structures, is also permitted in C language. Nested structures means, that one structure has another stucture as member variable.

Example:

```
struct Student

{

    char[30] name;

    int age;

    /* here Address is a structure */

    struct Address

    {

        char[50] locality;

        char[50] city;

        int pincode;

    }addr;

};
```

•

Structure as Function Arguments

We can pass a structure as a function argument just like we pass any other variable or an array as a function argument.

Example:

```
#include<stdio.h>

struct Student
{
    char name[10];
    int roll;
};

void show(struct Student st);

void main()
{
    struct Student std;
```

•

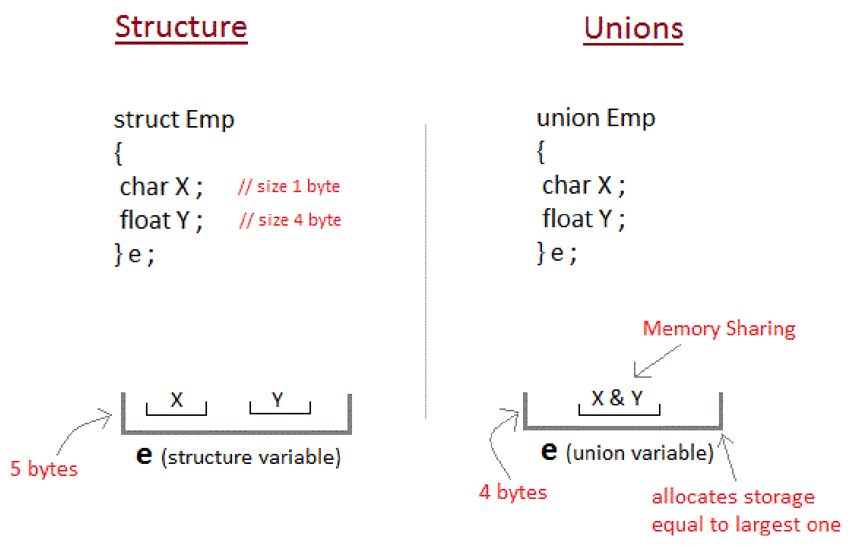
```
    printf("\nEnter Student record:\n");

    printf("\nStudent name:\t");

    scanf("%s", std.name);

    printf("\nEnter Student rollno.:\t");

    scanf("%d", &std.roll);

    show(std);

}


void show(struct Student st)

{

    printf("\nstudent name is %s", st.name);

    printf("\nroll is %d", st.roll);

}
```

Unions in C Language

•

Unions are conceptually similar to **structures**. The syntax to declare/define a union is also similar to that of a structure.

The only differences is in terms of storage. In **structure** each member has its own storage location, whereas all members of **union** uses a single shared memory location which is equal to the size of its largest data member.



This implies that although a union may contain many members of different types, it cannot handle all the members at the same time. A union is declared using the union keyword.

```
union item

{

   int m;

   float x;

   char c;
```

.

```
}It1;
```

This declares a variable It1 of type union item. This union contains three members each with a different data type. However only one of them can be used at a time. This is due to the fact that only one location is allocated for all the union variables, irrespective of their size.

The compiler allocates the storage that is large enough to hold the largest variable type in the union.

In the union declared above the member x requires 4 bytes which is largest amongst the members for a 16-bit machine. Other members of union will share the same memory address.

Accessing a Union Member

Syntax for accessing any union member is similar to accessing structure members,

```
union test
{
   int a;
   float b;
   char c;
}t;
```

•

```
t.a;    //to access members of union t

t.b;

t.c;
```

Example

```
#include <stdio.h>

union im

{

    int a;

    float b;

    char ch;

};


int main( )

{

    union item it;

    it.a = 12;
```

.

```
    it.b = 20.2;

    it.ch = 'z';



    printf("%d\n", it.a);

    printf("%f\n", it.b);

    printf("%c\n", it.ch);



    return 0;

}
```

OUTPUT:

```
-26426

20.1999

z
```

As you can see here, the values of a and b get corrupted and only variable c prints the expected result. This is because in union, the memory is shared among different data types. Hence, the only member whose value is currently stored will have the memory.

•

In the above example, value of the variable c was stored at last, hence the value of other variables is lost.

.

MODULE 3

Introduction to Pointers

- A Pointer in C language is a variable which holds the address of another variable of same data type.
- Pointers are used to access memory and manipulate the address.
- Pointers are one of the most distinct and exciting features of C language. It provides power and flexibility to the language.
- Whenever a variable is defined in C language, a memory location is assigned for it, in which its value will be stored. We can easily check this memory address, using the & symbol.
- If var is the name of the variable, then &var will give it's address.

```
#include<stdio.h>
void main()
{
    int var = 7;
    printf("Value of the variable var is: %d\n", var);
    printf("Memory address of the variable var is: %x\n", &var);
}
```

OUTPUT:

.

```
Value of the variable var is: 7
Memory address of the variable var is: bcc7a00
```

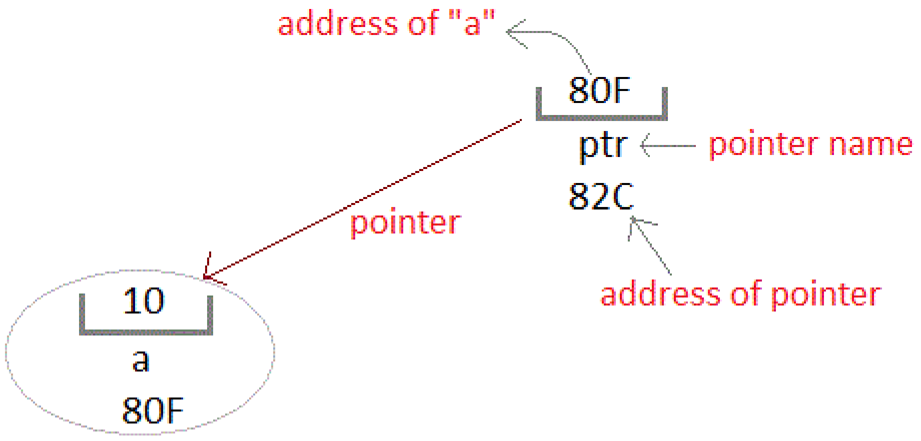
Concept of Pointers

- Whenever a variable is declared in a program, system allocates a location i.e an address to that variable in the memory, to hold the assigned value.
- This location has its own address number, which we just saw above.
- Let us assume that system has allocated memory location 80F for a variable a.

```
int a = 10;
```

- We can access the value 10 either by using the variable name a or by using its address 80F.
- The variables which are used to hold memory addresses are called Pointer variables.
- A pointer variable is therefore nothing but a variable which holds an address of some other variable.
- And the value of a pointer variable gets stored in another memory location.

•



Benefits of using pointers

a) Pointers are more efficient in handling Arrays and Structures.
b) Pointers allow references to function and thereby helps in passing of function as arguments to other functions.
c) It reduces length of the program and its execution time as well.
d) It allows C language to support Dynamic Memory management.

Dynamic Memory Allocation

- **Dynamic Memory Allocation** can be defined as a procedure in which the size of a data structure (like Array) is changed during the runtime.
- C provides some functions to achieve these tasks. There are 4 library functions provided by C defined under **<stdlib.h>** header file to facilitate dynamic memory allocation in C programming. They are:

•

1. malloc(): **“malloc”** or **“memory allocation”** method is used to dynamically allocate a single large block of memory with the specified size.
2. calloc():**“calloc”** or **“contiguous allocation”** method is used to dynamically allocate the specified number of blocks of memory of the specified type
3. free():**“free”** method is used to dynamically **de-allocate** the memory. The memory allocated using functions malloc() and calloc() are not de-allocated on their own.
4. realloc():**“realloc”** or **“re-allocation”** method is used to dynamically change the memory allocation of a previously allocated memory.

Declaring, Initializing and using a pointer variable

Declaration of Pointer variable

- General syntax of pointer declaration is,

```
datatype *pointer_name;
```

- Data type of a pointer must be same as the data type of the variable to which the pointer variable is pointing.
- void type pointer works with all data types, but is not often used.
- Here are a few examples:

```
int *ip;     // pointer to integer variable
```

•

```
float *fp;      // pointer to float variable

double *dp;     // pointer to double variable

char *cp;       // pointer to char variable
```

Initialization of Pointer variable

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

```
#include<stdio.h>

void main()

{

     int a = 10;

    int *ptr;      //pointer declaration

     ptr = &a;      //pointer initialization

}
```

•

- Pointer variables always point to variables of same data-type. Let's have an example to showcase this:

```
#include<stdio.h>
void main()
{
           float a;
           int *ptr;
          ptr = &a;      // ERROR, type mismatch
}
```

- If you are not sure about which variable's address to assign to a pointer variable while declaration, it is recommended to assign a NULL value to your pointer variable.
- A pointer which is assigned a NULLvalue is called a NULL pointer.

```
          #include <stdio.h>
          int main()
          {
                     int *ptr = NULL;
```

•

```
                    return 0;
              }
```

Using the pointer or Dereferencing of Pointer

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

```
#include <stdio.h>


int main()
{
  int a, *p;  // declaring the variable and pointer
  a = 10;
  p = &a;     // initializing the pointer
  printf("%d", *p);    //this will print the value of 'a'
  printf("%d", *&a);   //this will also print the value of 'a'
  printf("%u", &a);    //this will print the address of 'a'
  printf("%u", p);     //this will also print the address of 'a'
  printf("%u", &p);    //this will print the address of 'p'
  return 0;
}
```

•

Points to remember while using pointers:

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

Example

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

•

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


         /*
           below, '*a' is read as 'value at a'
           which is 10
       */
       printf("Value at the address, which is stored by pointer variable
                                                a is %d\n", *a);
       return 0;
}
```

OUTPUT:

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

Pointer to a Pointer (Double Pointer)

•

- Pointers are used to store the address of other variables of similar data-type. But if you want to store the address of a pointer variable, then you again need a pointer to store it.
- Thus, when one pointer variable stores the address of another pointer variable, it is known as Pointer to Pointer variable or Double Pointer.

Syntax:

```
int **p1;
```

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

```
int ***p2
```

Simple program to represent Pointer to a Pointer

```
#include <stdio.h>

int main() {

        int  a = 10;

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

        int  **p2;

        /*

        this can store the address of pointer variable p1 only.

        It cannot store the address of variable 'a'
```

•

```
        */
        p1 = &a;
        p2 = &p1;
        printf("Address of a = %u\n", &a);
        printf("Address of p1 = %u\n", &p1);
        printf("Address of p2 = %u\n\n", &p2);
        // below print statement will give the address of 'a'
        printf("Value at the address stored by p2 = %u\n", *p2);
        printf("Value at the address stored by p1 = %d\n\n", *p1);
        printf("Value of **p2 = %d\n", **p2); //read this *(*p2)
        /*
         This is not allowed, it will give a compile time error-
        p2 = &a;
        printf("%u", p2);
        */
        return 0;
}
```

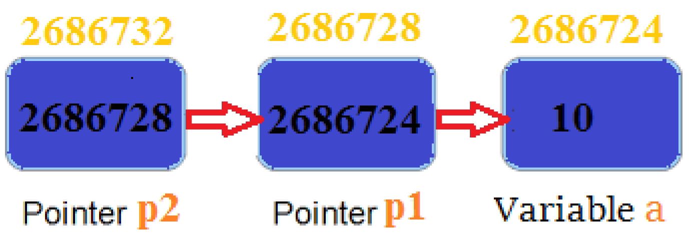
OUTPUT:

```
        Address of a = 2686724
        Address of p1 = 2686728
```

•

```
Address of p2 = 2686732
Value at the address stored by p2 = 2686724
Value at the address stored by p1 = 10
Value of **p2 = 10
```

Explanation of the above program:



- p1 pointer variable can only hold the address of the variable a (i.e Number of indirection operator(*)-1 variable). Similarly, p2 variable can only hold the address of variable p1. It cannot hold the address of variable a.
- *p2 gives us the value at an address stored by the p2 pointer. p2 stores the address of p1pointer and value at the address of p1 is the address of variable a. Thus, *p2 prints address of a.
- **p2 can be read as *(*p2). Hence, it gives us the value stored at the address *p2. From above statement, you know *p2 means the address of variable a. Hence, the value at the address *p2 is 10. Thus, **p2 prints 10.

Pointer and Arrays

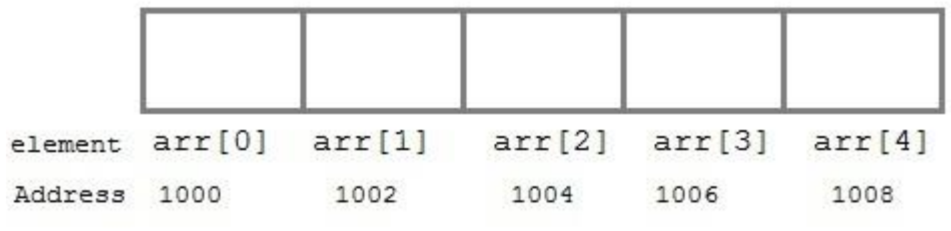
-

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

Suppose we declare an array arr,

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

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



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

arr is equal to &arr[0] by default

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

int *p;

•

```
        p = arr;

        // or,

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

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

Pointer to Array

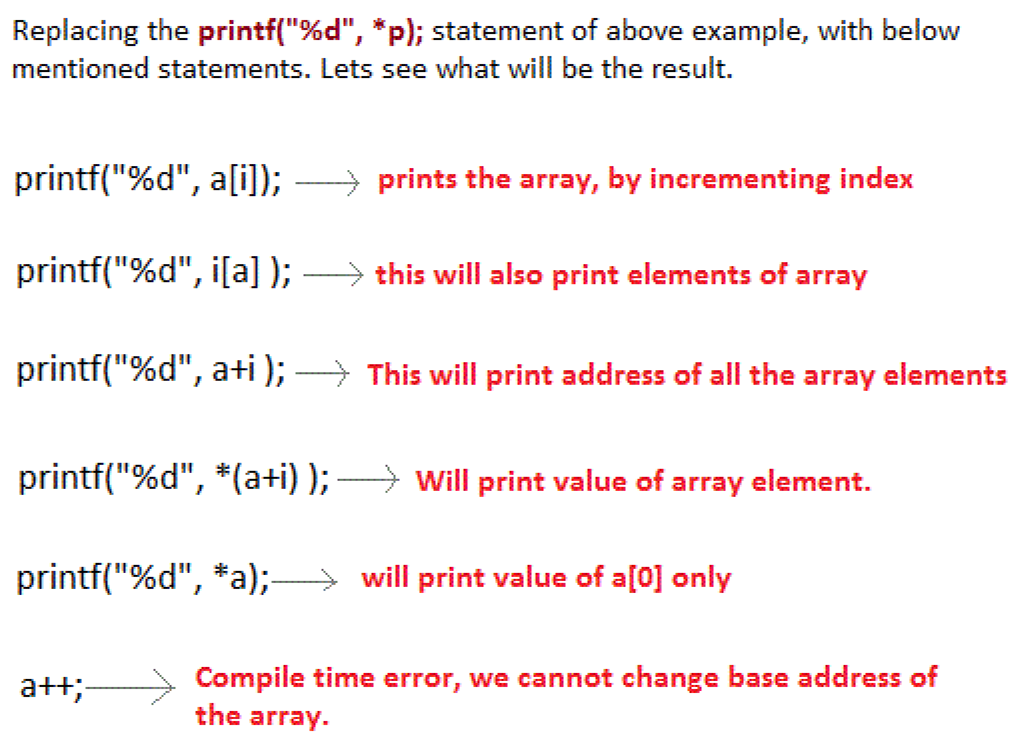
- As studied above, we can use a pointer to point to an array, and then we can use that pointer to access the array elements. Lets have an example,

```
#include <stdio.h>

int main()

{

        int i;

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

        int *p = a;     // same as int*p = &a[0]

        for (i = 0; i < 5; i++)

        {

                printf("%d", *p);

                p++;

        }
```

•

```
        return 0;
}
```

- In the above program, the pointer *p will print all the values stored in the array one by one. We can also use the Base address (a in above case) to act as a pointer and print all the values.



Replacing the **printf("%d", *p);** statement of above example, with below mentioned statements. Lets see what will be the result.

printf("%d", a[i]); ⟶ **prints the array, by incrementing index**

printf("%d", i[a]); ⟶ **this will also print elements of array**

printf("%d", a+i); ⟶ **This will print address of all the array elements**

printf("%d", *(a+i)); ⟶ **Will print value of array element.**

printf("%d", *a); ⟶ **will print value of a[0] only**

a++; ⟶ **Compile time error, we cannot change base address of the array.**

- The generalized form for using pointer with an array,

  *(a+i)

.

is same as:

a[i]

Pointer to Multidimensional Array

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

  ((a + i) + j)

  which is same as,

  a[i][j]

Pointer and Character strings

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

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

- The above code creates a string and stores its address in the pointer variable str.
- The pointer str now points to the first character of the string "Hello".

•

- Another important thing to note here is that the string created using char pointer can be assigned a value at runtime.

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

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

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

puts(str);
```

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

Array of Pointers

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

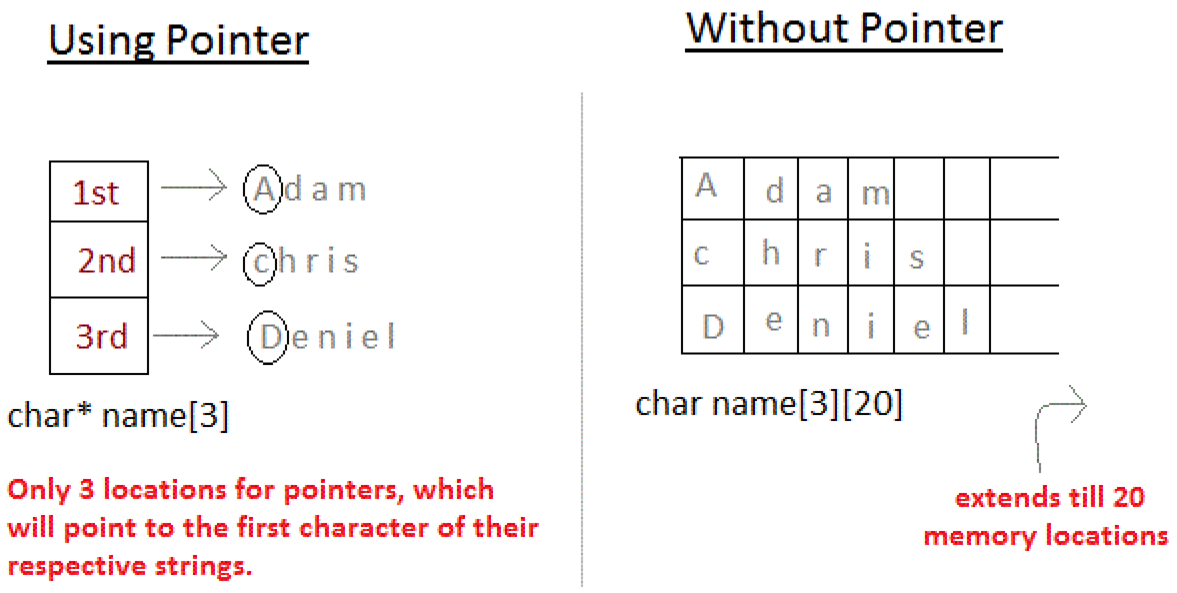
```
char *name[3] = {
                    "Adam",
                   "chris",
                   "Deniel"
                 };
```

//Now lets see same array without using pointer

```
char name[3][20] = {
```

•

```
                        "Adam",
                        "chris",
                        "Deniel"
                    };
```


Using Pointer
Without Pointer
1st
2nd
3rd
Adam
chris
Deniel
char* name[3]
Only 3 locations for pointers, which will point to the first character of their respective strings.
A d a m
c h r i s
D e n i e l
char name[3][20]
extends till 20 memory locations

- In the second approach memory wastage is more, hence it is prefered to use pointer in such cases.
- When we say memory wastage, it doesn't means that the strings will start occupying less space, no, characters will take the same space, but when we define array of characters, a contiguous memory space is located equal to the

•

maximum size of the array, which is a wastage, which can be avoided if we use pointers instead.

PROGRAMS WITH POINTERS

Program to access elements of an array using pointers:

```
#include<stdio.h>
#include<conio.h>
#define MAX 30

void main() {
   int size, i, arr[MAX];
   int *ptr;
   clrscr();

   ptr = &arr[0];

   printf("\nEnter the size of array : ");
   scanf("%d", &size);

   printf("\nEnter %d integers into array: ", size);
   for (i = 0; i < size; i++) {
      scanf("%d", ptr);
```

•

```
    ptr++;
  }

  printf("\nElements of array  are :");

  for (i = 0; i <size; i++) {
    printf("\nElement%d is %d : ", i, *ptr);
    ptr--;
  }
  getch();

}
```

Program to access elements of an array in reverse order using pointers:

```
#include<stdio.h>
#include<conio.h>
#define MAX 30

void main() {
  int size, i, arr[MAX];
  int *ptr;
  clrscr();

  ptr = &arr[0];

  printf("\nEnter the size of array : ");
  scanf("%d", &size);

  printf("\nEnter %d integers into array: ", size);
  for (i = 0; i < size; i++) {
    scanf("%d", ptr);
```

•

```
    ptr++;
  }

  ptr = &arr[size - 1];

  printf("\nElements of array in reverse order are :");

  for (i = size - 1; i >= 0; i--) {
    printf("\nElement%d is %d : ", i, *ptr);
    ptr--;
  }

  getch();
}
```

Program to sort elements of an array using pointers:

```
#include <stdio.h>
void main()
{
  int *a,i,j,tmp,n;
        printf("\n\n Pointer : Sort an array using pointer :\n");
        printf("--------------------------------------------\n");

  printf(" Input the number of elements to store in the array : ");
  scanf("%d",&n);

  printf(" Input %d number of elements in the array : \n",n);
  for(i=0;i<n;i++)
```

•

```
    {
        printf(" element - %d : ",i+1);
        scanf("%d",a+i);
        }
  for(i=0;i<n;i++)
  {
   for(j=i+1;j<n;j++)
   {
     if( *(a+i) > *(a+j))
     {
    tmp = *(a+i);
    *(a+i) = *(a+j);
    *(a+j) = tmp;
     }
   }
  }
  printf("\n The elements in the array after sorting : \n");
  for(i=0;i<n;i++)
    {
        printf(" element - %d : %d \n",i+1,*(a+i));
        }
printf("\n");
}
```

Program to compute sum of the array elements using pointers:

```
#include<stdio.h>
#include<conio.h>
void main() {
  int arr[10];
```

•

```
   int i, sum = 0;
   int *ptr;

   printf("\nEnter 10 elements : ");

   for (i = 0; i < 10; i++)
      scanf("%d", &arr[i]);

   ptr = arr; /* a=&a[0] */

   for (i = 0; i < 10; i++) {
      sum = sum + *ptr;
      ptr++;
   }

   printf("The sum of array elements : %d", sum);
}
```

Program to store information and display it using structure:

```
#include <stdio.h>
struct student
{
```

```
•

    char name[50];
    int roll;
    float marks;
} s;

int main()
{
    printf("Enter information:\n");
    printf("Enter name: ");
    scanf("%s", s.name);
    printf("Enter roll number: ");
    scanf("%d", &s.roll);
    printf("Enter marks: ");
    scanf("%f", &s.marks);

    printf("Displaying Information:\n");
    printf("Name: ");
    puts(s.name);
 printf("Roll number: %d\n",s.roll);
printf("Marks: %.1f\n", s.marks);
```

•

```
return 0;

}
```

Program to add two numbers using pointers:

```
#include<stdio.h>

int main() {
   int *ptr1, *ptr2;
   int num;

   printf("\nEnter two numbers : ");
   scanf("%d %d", ptr1, ptr2);

   num = *ptr1 + *ptr2;

   printf("Sum = %d", num);
   return (0);
}
```

Program to swap two numbers using pointers:

```
#include <stdio.h>

int main()
{
   int x, y, *a, *b, temp;
```

•

```
   printf("Enter the value of x and y\n");
   scanf("%d%d", &x, &y);

   printf("Before Swapping\nx = %d\ny = %d\n", x, y);

   a = &x;
   b = &y;

   temp = *b;
   *b = *a;
   *a = temp;

   printf("After Swapping\nx = %d\ny = %d\n", x, y);

   return 0;
}
```

Program to concatenate two strings using pointers:

```
#include <stdio.h>

int main()
{
    char aa[100], bb[100];

    printf("\nEnter the first string: ");
    gets(aa);   // inputting first string

    printf("\nEnter the second string to be concatenated: ");
    gets(bb);   // inputting second string
```

•

```
    char *a = aa;
    char *b = bb;

    // pointing to the end of the 1st string
    while(*a)   // till it doesn't point to NULL-till string is not empty
    {
        a++;    // point to the next letter of the string
    }

    while(*b)   // till second string is not empty
    {
        *a = *b;
        b++;
        a++;
    }

    *a = '\0';  // string must end with '\0'

    printf("\n\n\nThe string after concatenation is: %s ", aa);
    return 0;
}
```

Program to compare two strings using pointers:

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

    char string1[50],string2[50],*str1,*str2;
    int i,equal = 0;
```

•

```
   printf("Enter The First String: ");
   scanf("%s",string1);

   printf("Enter The Second String: ");
   scanf("%s",string2);

   str1 = string1;
   str2 = string2;
   while(*str1 == *str2)
   {

      if ( *str1 == '\0' || *str2 == '\0' )
         break;

      str1++;
      str2++;

   }

   if( *str1 == '\0' && *str2 == '\0' )
      printf("\n\nBoth Strings Are Equal.");

   else
      printf("\n\nBoth Strings Are Not Equal.");

}
```

Program to copy a string to another string using pointers:

•

```
#include<stdio.h>
void main()
{
             char*str1="Hello world";
             char str2[30];
             clrscr();
            while(*str1!='\0')
                *str2++=*str1++;
            *str2='\0';
             printf("\n %s",str2);
             getch();
 }
```

Program to find the length of a string using pointers:

```
#include<stdio.h>

int main() {

  char str[20], *pt;

  int i = 0;

  printf("Pointer Example Program : Find or Calculate Length of String \n");

  printf("Enter Any string [below 20 chars] : ");

  gets(str);

  pt = str;

  while (*pt != '\0') {

    i++;
```

•

```
    pt++;
  }
  printf("Length of String : %d", i);
  return 0;
}
```

.

MODULE 4

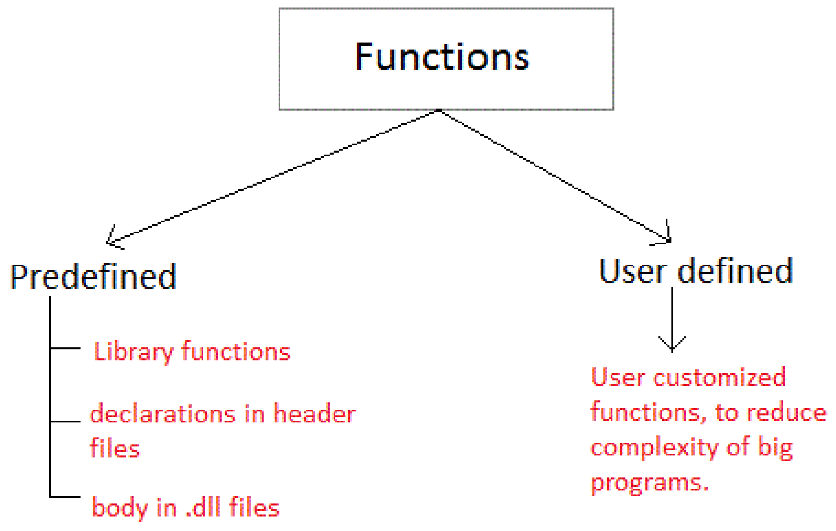
Functions in C

- A function is a block of code that performs a particular task.
- There are many situations where we might need to write same line of code for more than once in a program.
- This may lead to unnecessary repetition of code, bugs and even becomes boring for the programmer.
- So, C language provides an approach in which you can declare and define a group of statements once in the form of a function and it can be called and used whenever required.
- These functions defined by the user are also known as User-defined Functions
- C functions can be classified into two categories,

  Library functions

  User-defined functions

•


Functions
Predefined
User defined
Library functions
declarations in header files
body in .dll files
User customized functions, to reduce complexity of big programs.

- Library functions are those functions which are already defined in C library, example printf(), scanf(), strcat() etc. You just need to include appropriate header files to use these functions. These are already declared and defined in C libraries.
- A User-defined functions on the other hand, are those functions which are defined by the user at the time of writing program. These functions are made for code reusability and for saving time and space.

Benefits of Using Functions

- It provides modularity to your program's structure.
- It makes your code reusable. You just have to call the function by its name to use it, wherever required.

.

- In case of large programs with thousands of code lines, debugging and editing becomes easier if you use functions.
- It makes the program more readable and easy to understand.

Function Declaration

- General syntax for function declaration is,

```
returntype functionName(type1 parameter1, type2 parameter2,...);
```

- Like any variable or an array, a function must also be declared before it's used.
- Function declaration informs the compiler about the function name, parameters is accept, and its return type.
- The actual body of the function can be defined separately. It's also called as Function Prototyping.

- Function declaration consists of 4 parts.

    returntype

    function name

    parameter list

    terminating semicolon

returntype

•

- When a function is declared to perform some sort of calculation or any operation and is expected to provide with some result at the end, in such cases, a return statement is added at the end of function body.
- Return type specifies the type of value (int, float, char, double) that function is expected to return to the program which called the function.

Note: In case your function doesn't return any value, the return type would be void.

functionName

- Function name is an identifier and it specifies the name of the function.
- The function name is any valid C identifier and therefore must follow the same naming rules like other variables in C language.

parameter list

- The parameter list declares the type and number of arguments that the function expects when it is called.
- Also, the parameters in the parameter list receives the argument values when the function is called. They are often referred as formal parameters.

Example

Let's write a simple program with a main() function, and a user defined function to multiply two numbers, which will be called from the main() function.

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

•

```
int multiply(int a, int b);     // function declaration

int main()
{
    int i, j, result;
    printf("Please enter 2 numbers you want to multiply...");
    scanf("%d%d", &i, &j);
    result = multiply(i, j);        // function call
    printf("The result of muliplication is: %d", result);
    return 0;
}

int multiply(int a, int b)
{
    return (a*b);       // function defintion, this can be done in one line
}
```

•

Function definition Syntax

- Just like in the example above, the general syntax of function definition is,

```
returntype functionName(type1 parameter1, type2 parameter2,...)
{
        // function body goes here
}
```

- The first line returntype functionName(type1 parameter1, type2 parameter2,...) is known as function header and the statement(s) within curly braces is called function body.

functionbody

- The function body contains the declarations and the statements(algorithm) necessary for performing the required task.
- The body is enclosed within curly braces { ... } and consists of three parts.

  (a) Local variable declaration (if required).

  (b) Function statements to perform the task inside the function.

  (c) A return statement to return the result evaluated by the function (if return type is void, then no return statement is required).

•

Calling a function

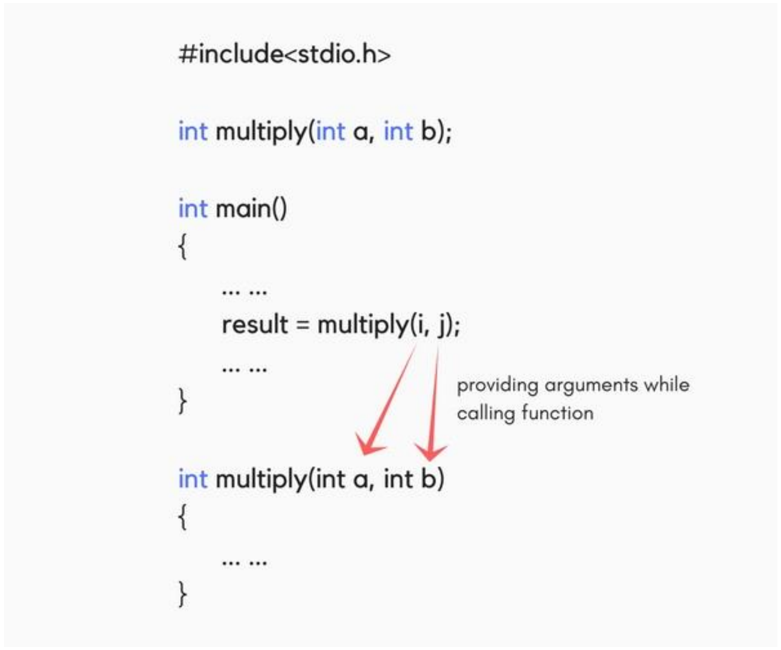
- When a function is called, control of the program gets transferred to the function.

functionName(argument1, argument2,...);

Passing Arguments to a function

- Arguments are the values specified during the function call, for which the formal parameters are declared while defining the function.

- It is possible to have a function with parameters but no return type. It is not necessary, that if a function accepts parameter(s), it must return a result too.



```
#include<stdio.h>

int multiply(int a, int b);

int main()
{
    ... ...
    result = multiply(i, j);
    ... ...
}

int multiply(int a, int b)
{
    ... ...
}
```

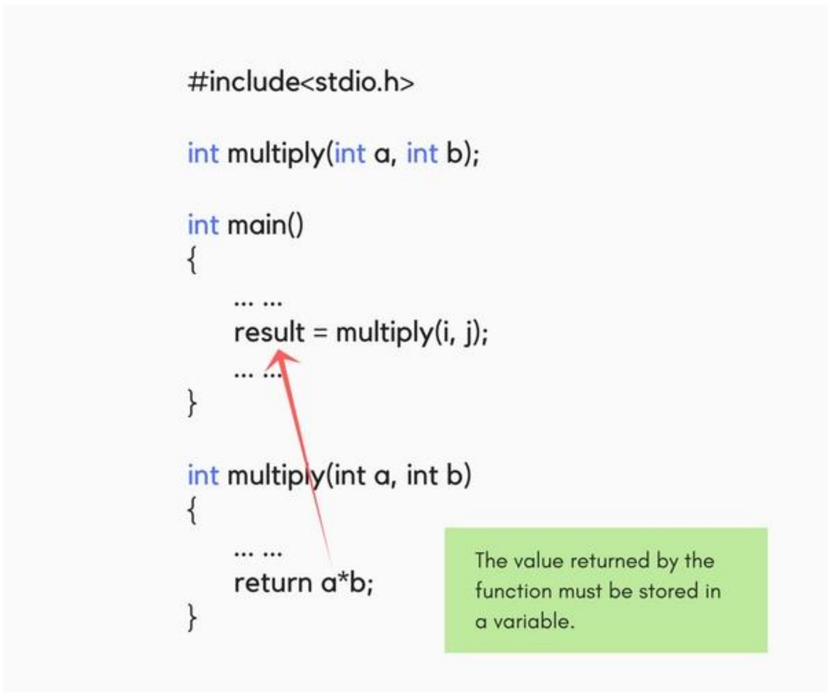
•

- While declaring the function, we have declared two parameters a and b of type int. Therefore, while calling that function, we need to pass two arguments, else we will get compilation error.
- And the two arguments passed should be received in the function definition, which means that the function header in the function definition should have the two parameters to hold the argument values. T
- These received arguments are also known as formal parameters. The name of the variables while declaring, calling and defining a function can be different.

Returning a value from function

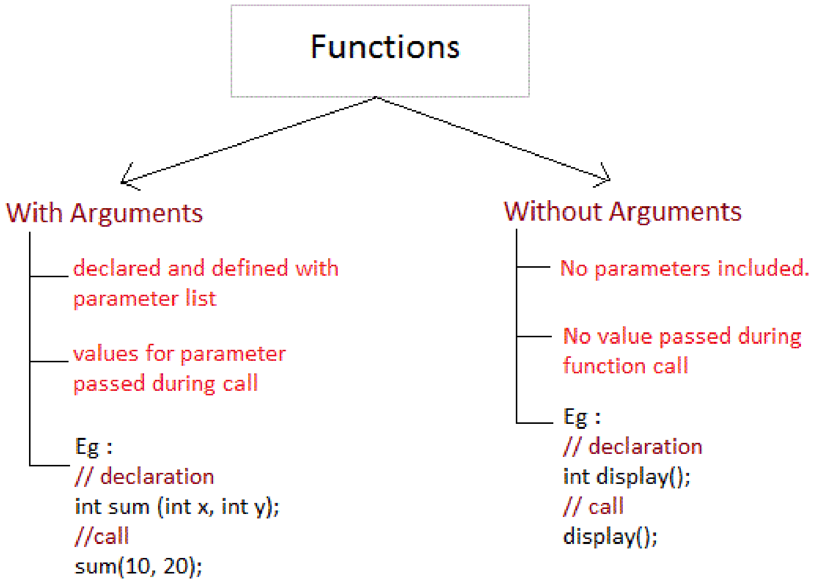
- A function may or may not return a result. But if it does, we must use the return statement to output the result. return statement also ends the function execution, hence it must be the last statement of any function.
- If you write any statement after the return statement, it won't be executed.

•



```
#include<stdio.h>

int multiply(int a, int b);

int main()
{
    ... ...
    result = multiply(i, j);
    ... ...
}

int multiply(int a, int b)
{
    ... ...
    return a*b;
}
```

The value returned by the function must be stored in a variable.



• The datatype of the value returned using the return statement should be same as the return type mentioned at function

•

declaration and definition. If any of it mismatches, you will get compilation error.

Type of User-defined Functions in C

- There can be 4 different types of user-defined functions, they are:

  Function with no arguments and no return value

  Function with no arguments and a return value

  Function with arguments and no return value

  Function with arguments and a return value

Function with no arguments and no return value

- Such functions can either be used to display information or they are completely dependent on user inputs.
- Below is an example of a function, which takes 2 numbers as input from user, and display which is the greater number.

```
#include<stdio.h>
void greatNum();       // function declaration

int main()
{
        greatNum();        // function call
```

•

```
          return 0;
}

void greatNum()       // function definition
{
          int i, j;
          printf("Enter 2 numbers that you want to compare...");
          scanf("%d%d", &i, &j);
          if(i > j) {
                    printf("The greater number is: %d", i);
                       }
          else {
                    printf("The greater number is: %d", j);
                  }
}
```

Function with no arguments and a return value

- We have modified the above example to make the function greatNum() return the number which is greater amongst the 2 input numbers.

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

•

```
int greatNum();       // function declaration
int main()
{
        int result;
        result = greatNum();       // function call
        printf("The greater number is: %d", result);
        return 0;
}

int greatNum()       // function definition
{
        int i, j, greaterNum;
        printf("Enter 2 numbers that you want to compare...");
        scanf("%d%d", &i, &j);
        if(i > j) {
                greaterNum = i;
        }
        else {
                greaterNum = j;
        }
```

•

```
            // returning the result
            return greaterNum;
    }
```

Function with arguments and no return value

- This time, we have modified the above example to make the function greatNum() take two int values as arguments, but it will not be returning anything.

```
#include<stdio.h>
void greatNum(int a, int b);       // function declaration
int main()
{
        int i, j;
        printf("Enter 2 numbers that you want to compare...");
        scanf("%d%d", &i, &j);
        greatNum(i, j);        // function call
        return 0;
}
```

•

```
void greatNum(int x, int y)        // function definition
{
        if(x > y) {
                printf("The greater number is: %d", x);
        }
        else {
                printf("The greater number is: %d", y);
        }
}
```

Function with arguments and a return value

- This is the best type, as this makes the function completely independent of inputs and outputs, and only the logic is defined inside the function body.

```
#include<stdio.h>
int greatNum(int a, int b);       // function declaration
int main()
{
        int i, j, result;
        printf("Enter 2 numbers that you want to compare...");
        scanf("%d%d", &i, &j);
```

•

```
        result = greatNum(i, j); // function call
        printf("The greater number is: %d", result);
        return 0;
}

int greatNum(int x, int y)      // function definition
{
        if(x > y) {
                return x;
        }
        else {
                return y;
        }
}
```

Nesting of Functions

- C language also allows nesting of functions i.e to use/call one function inside another function's body.

```
function1()
{
```

•

```
        // function1 body here
        function2();
        // function1 body here      }
```

- If function2() also has a call for function1() inside it, then in that case, it will lead to an infinite nesting. They will keep calling each other and the program will never terminate.

What is Recursion?

- Recursion is a special way of nesting functions, where a function calls itself inside it.
- We must have certain conditions in the function to break out of the recursion, otherwise recursion will occur infinite times.

```
function1()
{
        // function1 body
        function1();
         // function1 body
}
```

Example: Factorial of a number using Recursion

•

```
#include<stdio.h>
int factorial(int x);       //declaring the function
void main()
{
    int a, b;
    printf("Enter a number...");
    scanf("%d", &a);
    b = factorial(a);       //calling the function named factorial
    printf("%d", b);
}

int factorial(int x) //defining the function
{
    int r = 1;
    if(x == 1)
        return 1;
    else
        r = x*factorial(x-1);       //recursion, since the function calls itself
    return r;
```

.

}

Types of Function calls in C

- In functions with arguments, we can call a function in two different ways, based on how we specify the arguments, and these two ways are:

    Call by Value

    Call by Reference

Call by Value

- Calling a function by value means, we pass the values of the arguments which are stored or copied into the formal parameters of the function.
- Hence, the original values are unchanged only the parameters inside the function changes.

```
#include<stdio.h>
void calc(int x);
int main()
{
        int x = 10;
        calc(x);
```

•

```
        // this will print the value of 'x'
        printf("\nvalue of x in main is %d", x);
        return 0;
}

void calc(int x)
{
        // changing the value of 'x'
        x = x + 10 ;
        printf("value of x in calc function is %d ", x);
}
```

OUTPUT:

```
        value of x in calc function is 20
        value of x in main is 10
```

- In this case, the actual variable x is not changed. This is because we are passing the argument by value, hence a copy of x is passed to the function, which is updated during function execution, and that copied value in the function is destroyed when the function ends(goes out of scope).
- So the variable x inside the main() function is never changed and hence, still holds a value of 10.

-

- But we can change this program to let the function modify the original x variable, by making the function calc() return a value, and storing that value in x.

```
#include<stdio.h>
int calc(int x);
int main()
{
        int x = 10;
        x = calc(x);
        printf("value of x is %d", x);
        return 0;
}

int calc(int x)
{
        x = x + 10 ;
        return x;
}
```

•

OUTPUT:

```
        value of x is 20
```

Call by Reference

- In call by reference we pass the address (reference) of a variable as argument to any function.
- When we pass the address of any variable as argument, then the function will have access to our variable, as it now knows where it is stored and hence can easily update its value.
- In this case the formal parameter can be taken as a reference or a pointer(don't worry about pointers, we will soon learn about them), in both the cases they will change the values of the original variable.

```
#include<stdio.h>
void calc(int *p);      // functin taking pointer as argument
int main()
{
        int x = 10;
        calc(&x);       // passing address of 'x' as argument
        printf("value of x is %d", x);
        return(0);
```

•

```
}



void calc(int *p)       //receiving the address in a reference pointer variable
{
        /*
        changing the value directly that is
        stored at the address passed
        */
        *p = *p + 10;
}
```

```
OUTPUT:
        value of x is 20
```

Pointers as Function Argument

- Pointer as a function parameter is used to hold addresses of arguments passed during function call. This is also known as call by reference.
- When a function is called by reference any change made to the reference variable will affect the original variable.

•

Example: Swapping two numbers using Pointer

```
#include <stdio.h>
void swap(int *a, int *b);
int main()
{
        int m = 10, n = 20;
        printf("m = %d\n", m);
        printf("n = %d\n\n", n);
        swap(&m, &n);    //passing address of m and n to the swap function
        printf("After Swapping:\n\n");
        printf("m = %d\n", m);
        printf("n = %d", n);
        return 0;
}

/*
        pointer 'a' and 'b' holds and
        points to the address of 'm' and 'n'
*/
```

•

```
void swap(int *a, int *b)
{
        int temp;
        temp = *a;
        *a = *b;
        *b = temp;
}
```

```
OUTPUT:
        m = 10
        n = 20
        After Swapping:
        m = 20
        n = 10
```

Functions returning Pointer variables

- A function can also return a pointer to the calling function. In this case you must be careful, because local variables of function doesn't live outside the function.

•

- They have scope only inside the function. Hence if you return a pointer connected to a local variable, that pointer will be pointing to nothing when the function ends.

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

•

OUTPUT:

```
92 is larger
```

Pointer to functions

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

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

Here is an example :

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

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

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

- Here s is a pointer to a function sum. Now sum can be called using function pointer s along with providing the required argument values.

•

s (10, 20);

Example of Pointer to Function

```
#include <stdio.h>
int sum(int x, int y)
{
        return x+y;
}

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

OUTPUT:

```
        25
```

.

MODULE 5
SORTING AND SEARCHING ALGORITHMS

Introduction to Sorting

- Sorting is nothing but arranging the data in ascending or descending order.
- Sorting arranges data in a sequence which makes searching easier.

Sorting Efficiency

- The two main criteria to judge which algorithm is better than the other have been:
    a) Time taken to sort the given data.
    b) Memory Space required to do so.

Bubble Sort Algorithm

- **Bubble Sort** is a simple algorithm which is used to sort a given set of n elements provided in form of an array with n number of elements.
- Bubble Sort compares all the element one by one and sort them based on their values.
- If the given array has to be sorted in ascending order, then bubble sort will start by comparing the first element of the array with the second element, if

•

the first element is greater than the second element, it will **swap** both the elements, and then move on to compare the second and the third element, and so on.

- If we have total n elements, then we need to repeat this process for n-1 times.
- It is known as **bubble sort**, because with every complete iteration the largest element in the given array, bubbles up towards the last place or the highest index, just like a water bubble rises up to the water surface.
- Sorting takes place by stepping through all the elements one-by-one and comparing it with the adjacent element and swapping them if required.

Implementing Bubble Sort Algorithm

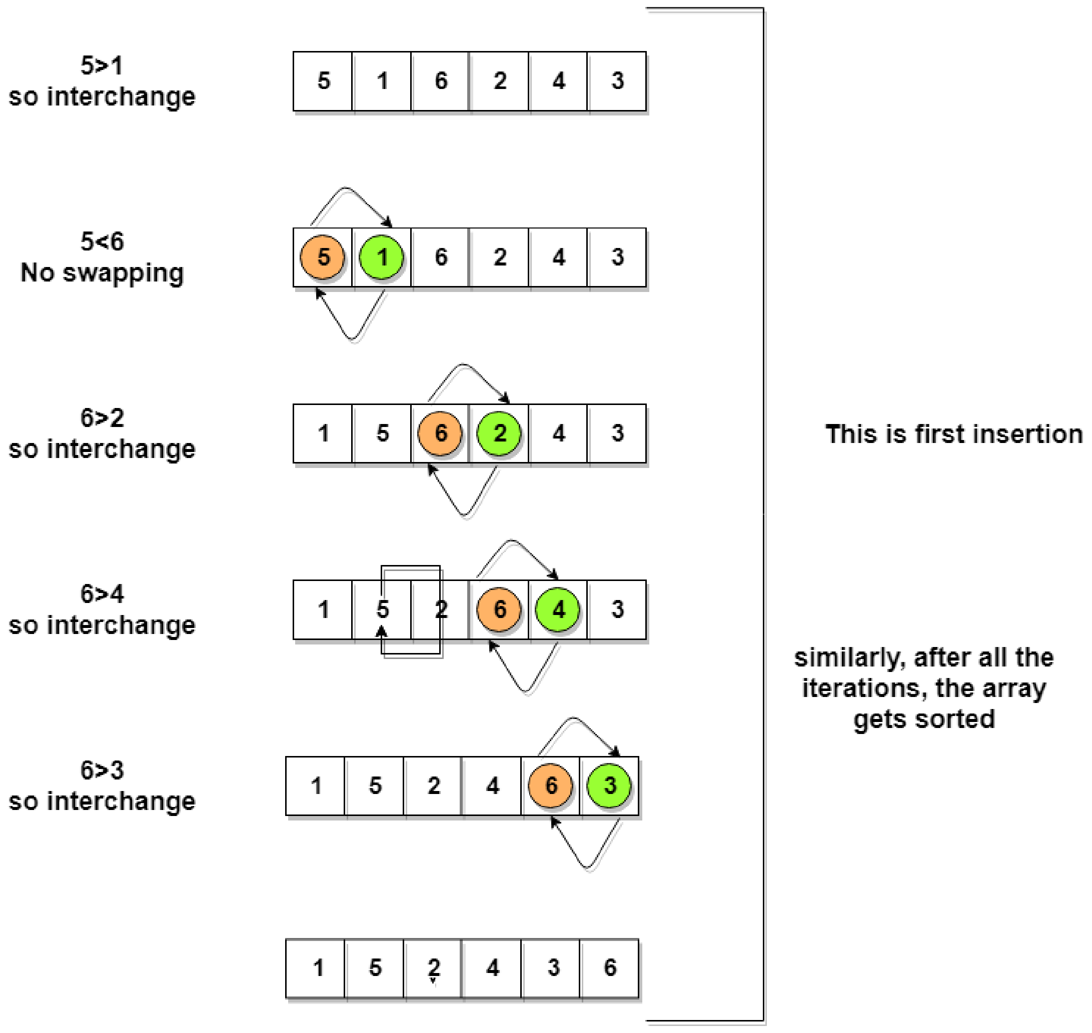
Following are the steps involved in bubble sort (for sorting a given array in ascending order):

a) Starting with the first element (index = 0), compare the current element with the next element of the array.
b) If the current element is greater than the next element of the array, swap them.
c) If the current element is less than the next element, move to the next element. Repeat Step 1.

Let's consider an array with values {5, 1, 6, 2, 4, 3}

Below, we have a pictorial representation of how bubble sort will sort the given array.

•


5>1
so interchange
5<6
No swapping
6>2
so interchange
6>4
so interchange
6>3
so interchange
This is first insertion
similarly, after all the
iterations, the array
gets sorted
5 1 6 2 4 3
5 1 6 2 4 3
1 5 6 2 4 3
1 5 2 6 4 3
1 5 2 4 6 3
1 5 2 4 3 6

•

So as we can see in the representation above, after the first iteration, 6 is placed at the last index, which is the correct position for it.

Similarly after the second iteration, 5 will be at the second last index, and so on.

```
// below we have a simple C program for bubble sort
#include <stdio.h>
void bubbleSort(int arr[], int n)
{
    int i, j, temp;
    for(i = 0; i < n; i++)
    {
        for(j = 0; j < n-i-1; j++)
        {
            if( arr[j] > arr[j+1])
            {
                // swap the elements
                temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
```

•

```
            }
        }
    }
    // print the sorted array
    printf("Sorted Array: ");
    for(i = 0; i < n; i++)
    {
        printf("%d  ", arr[i]);
    }
}

int main()
{
    int arr[100], i, n, step, temp;
    // ask user for number of elements to be sorted
    printf("Enter the number of elements to be sorted: ");
    scanf("%d", &n);
    // input elements if the array
    for(i = 0; i < n; i++)
    {
```

•

```
        printf("Enter element no. %d: ", i+1);
        scanf("%d", &arr[i]);
    }
    // call the function bubbleSort
    bubbleSort(arr, n);
    return 0;
}
```

Although the above logic will sort an unsorted array, still the above algorithm is not efficient because as per the above logic, the outer for loop will keep on executing for **6** iterations even if the array gets sorted after the second iteration.

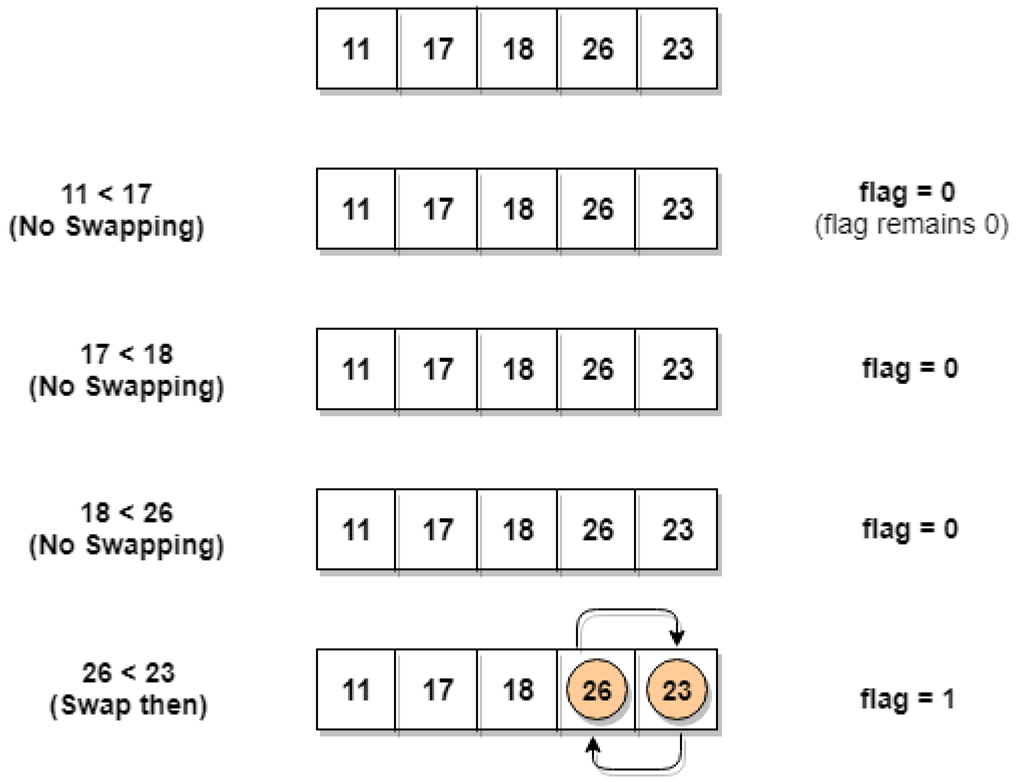
Optimized Bubble Sort Algorithm

- To optimize our bubble sort algorithm, we can introduce a flag to monitor whether elements are getting swapped inside the inner for loop.
- Hence, in the inner for loop, we check whether swapping of elements is taking place or not, everytime.
- If for a particular iteration, no swapping took place, it means the array has been sorted and we can jump out of the for loop, instead of executing all the iterations.

Let's consider an array with values {11, 17, 18, 26, 23}

•

Below, we have a pictorial representation of how the optimized bubble sort will sort the given array.



- As we can see, in the first iteration, swapping took place, hence we updated our flag value to 1, as a result, the execution enters the for loop again.
- But in the second iteration, no swapping will occur, hence the value of flag will remain 0, and execution will break out of loop.

// below we have a simple C program for bubble sort

•

```
#include <stdio.h>
void bubbleSort(int arr[], int n)
{
    int i, j, temp;
    for(i = 0; i < n; i++)
    {
        for(j = 0; j < n-i-1; j++)
        {
            // introducing a flag to monitor swapping
            int flag = 0;
            if( arr[j] > arr[j+1])
            {
                // swap the elements
                temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
                // if swapping happens update flag to 1
                flag = 1;
            }
        }
```

•

```
        // if value of flag is zero after all the iterations of inner loop
        // then break out
        if(!flag)
        {
            break;
        }
    }

    // print the sorted array
    printf("Sorted Array: ");
    for(i = 0; i < n; i++)
    {
        printf("%d  ", arr[i]);
    }
}

int main()
{
    int arr[100], i, n, step, temp;
    // ask user for number of elements to be sorted
```

•

```
    printf("Enter the number of elements to be sorted: ");
    scanf("%d", &n);
    // input elements if the array
    for(i = 0; i < n; i++)
    {
        printf("Enter element no. %d: ", i+1);
        scanf("%d", &arr[i]);
    }
    // call the function bubbleSort
    bubbleSort(arr, n);
    return 0;
}
```

In the above code, in the function bubbleSort, if for a single complete cycle of j iteration(inner for loop), no swapping takes place, then flag will remain 0 and then we will break out of the for loops, because the array has already been sorted.

Selection Sort Algorithm

- Selection sort is conceptually the simplest sorting algorithm.
- This algorithm will first find the **smallest** element in the array and swap it with the element in the **first** position, then it will find the **second smallest** element and swap it with the element in the **second** position, and it will keep on doing this until the entire array is sorted.

•

- It is called selection sort because it repeatedly **selects** the next-smallest element and swaps it into the right place.

How Selection Sort Works?

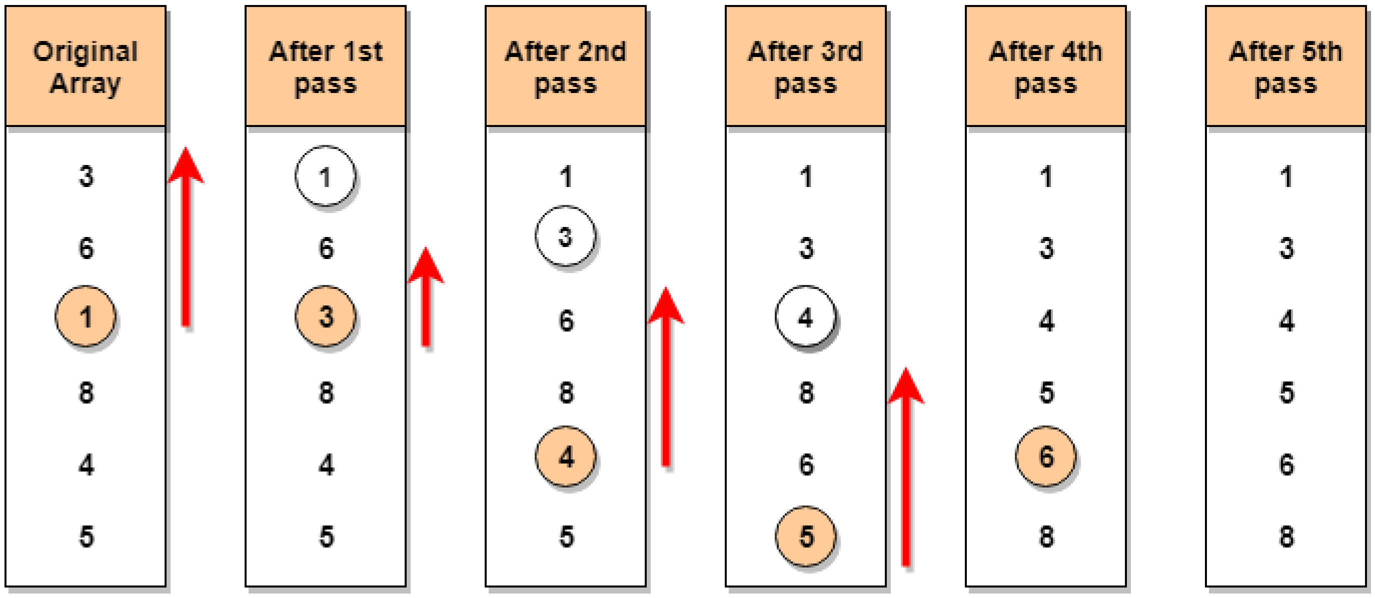
Following are the steps involved in selection sort (for sorting a given array in ascending order):

1. Starting from the first element, we search the smallest element in the array, and replace it with the element in the first position.
2. We then move on to the second position, and look for smallest element present in the subarray, starting from index 1, till the last index.
3. We replace the element at the **second** position in the original array, or we can say at the first position in the subarray, with the second smallest element.
4. This is repeated, until the array is completely sorted.

Let's consider an array with values {3,6,1,8,4,5}

•

Below, we have a pictorial representation of how selection sort will sort the given array.



- In the **first** pass, the smallest element will be 1, so it will be placed at the first position.
- Then leaving the first element, **next smallest** element will be searched, from the remaining elements. We will get 3 as the smallest, so it will be then placed at the second position.
- Then leaving 1 and 3 (because they are at the correct position), we will search for the next smallest element from the rest of the elements and put it at third position and keep doing this until array is sorted.

.

Finding Smallest Element in a subarray

- In selection sort, in the first step, we look for the smallest element in the array and replace it with the element at the first position.
- Consider that you have an array with following values {3, 6, 1, 8, 4, 5}. Now as per selection sort, we will start from the first element and look for the smallest number in the array, which is `1` and we will find it at the **index** `2`.
- Once the smallest number is found, it is swapped with the element at the first position.
- Well, in the next iteration, we will have to look for the second smallest number in the array.
- If you look closely, we already have the smallest number/element at the first position, which is the right position for it and we do not have to move it anywhere now.
- So we can say, that the first element is sorted, but the elements to the right, starting from index 1 are not.
- So, we will now look for the smallest element in the subarray, starting from index 1, to the last index.
- After we have found the second smallest element and replaced it with element on index 1(which is the second position in the array), we will have the first two positions of the array sorted.

•

- Then we will work on the subarray, starting from index 2 now, and again looking for the smallest element in this subarray.

Implementing Selection Sort Algorithm

- In the C program below, we have tried to divide the program into small functions, so that it's easier fo you to understand which part is doing what.

```
#include <stdio.h>
void swap(int *xp, int *yp)
{
   int temp = *xp;
   *xp = *yp;
   *yp = temp;
}

void selectionSort(int arr[], int n)
{
   int i, j, min_idx;

   // One by one move boundary of unsorted subarray
   for (i = 0; i < n-1; i++)
   {
      // Find the minimum element in unsorted array
      min_idx = i;
```

•

```
        for (j = i+1; j < n; j++)
          if (arr[j] < arr[min_idx])
            min_idx = j;

        // Swap the found minimum element with the first element
        swap(&arr[min_idx], &arr[i]);
    }
}
/* Function to print an array */
void printArray(int arr[], int size)
{
    int i;
    for (i=0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

// Driver program to test above functions
int main()
{
    int arr[],n;
// ask user for number of elements to be sorted
    printf("Enter the number of elements to be sorted: ");
    scanf("%d", &n);

    // input elements if the array
```

•

```
    for(i = 0; i < n; i++)
    {
        printf("Enter element no. %d: ", i+1);
        scanf("%d", &arr[i]);
    }
    selectionSort(arr, n);
    printf("Sorted array: \n");
    printArray(arr, n);
    return 0;
}
```

Introduction to Searching Algorithms

- To search an element in a given array, there are two popular algorithms available:
    a) Linear Search
    b) Binary Search

Linear Search

- Linear search is a very basic and simple search algorithm. In Linear search, we search an element or value in a given array by traversing the array from the starting, till the desired element or value is found.
- It compares the element to be searched with all the elements present in the array and when the element is matched successfully, it returns the index of the element in the array, else it return -1.

•

- Linear Search is applied on unsorted or unordered lists, when there are fewer elements in a list.

Implementing Linear Search

1. Traverse the array using a for loop.
2. In every iteration, compare the target value with the current value of the array.
    - If the values match, return the current index of the array.
    - If the values do not match, move on to the next array element.
3. If no match is found, return -1.

```
#include<stdio.h>
int linearSearch(int values[], int target, int n)
{
    for(int i = 0; i < n; i++)
    {
        if (values[i] == target)
        {
            return i;
        }
    }
    return -1;
```

•

```
int main(void)
{
  int values[],n,target,i;
   printf("Enter the number of elements: ");
   scanf("%d", &n);
   // input elements if the array
   for(i = 0; i < n; i++)
   {
      printf("Enter element no. %d: ", i+1);
      scanf("%d", &values[i]);
   }
   printf("Enter the element to be searched: ");
   scanf("%d", &target);

   int result = linearSearch (values, target,n);
   if(result == -1)
   {
      printf("Element is not present in the given array.");
   }
   else
   {
      printf("Element is present at index: %d", result);
   }
   return 0;
```

•

```
}
```

Features of Linear Search Algorithm

a) It is used for unsorted and unordered small list of elements.
b) It has a time complexity of O(n), which means the time is linearly dependent on the number of elements, which is not bad, but not that good too.
c) It has a very simple implementation.

Binary Search

- Binary Search is applied on the sorted array or list of large size. It's time complexity of O(log n) makes it very fast as compared to other sorting algorithms.
- The only limitation is that the array or list of elements must be sorted for the binary search algorithm to work on it.

Implementing Binary Search Algorithm

1. Start with the middle element:

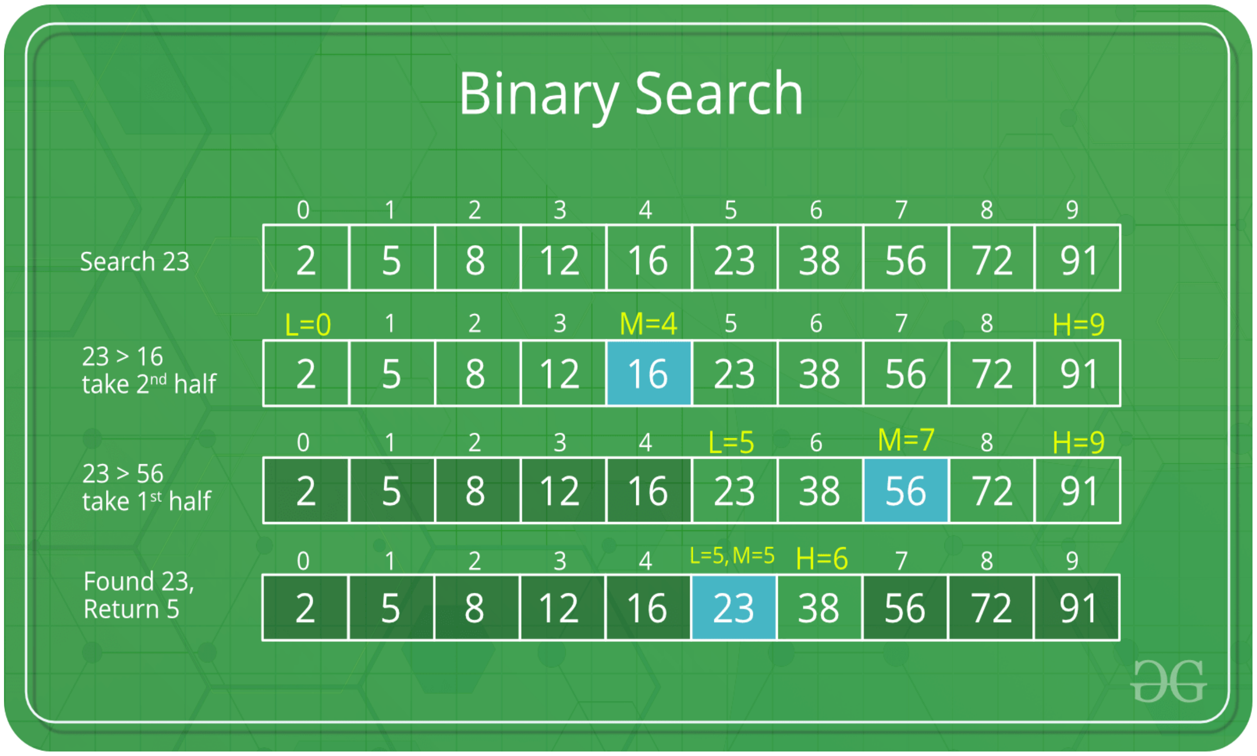
  - If the target value is equal to the middle element of the array, then return the index of the middle element.

•

- If not, then compare the middle element with the target value,
  - If the target value is greater than the number in the middle index, then pick the elements to the right of the middle index, and start with Step 1.
  - If the target value is less than the number in the middle index, then pick the elements to the left of the middle index, and start with Step 1.

2. When a match is found, return the index of the element matched.

3. If no match is found, then return -1



#include<stdio.h>

•

```
int binarySearch(int values[], int len, int target)
{
    int max = (len - 1);
    int min = 0;
    int guess;  // this will hold the index of middle elements
    while(max >= min)
    {
        guess = (max + min) / 2;

        if(values[guess] ==  target)
        {
            printf("Number of steps required for search: %d \n", step);
            return guess;
        }
        else if(values[guess] >  target)
        {
            // target would be in the left half
```

•

```
          max = (guess - 1);
        }
        else
        {
          // target would be in the right half
          min = (guess + 1);
        }
    }
    // We reach here when element is not
    // present in array
    return -1;
}
int main(void)
{
   int values[100],n,target,i;
    printf("Enter the number of elements: ");
    scanf("%d", &n);
```

•

```
// input elements if the array

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

{

    printf("Enter element no. %d: ", i+1);

    scanf("%d", &values[i]);

}

printf("Enter the element to be searched: ");

scanf("%d", &target);

int result = binarySearch(values, n, target);

if(result == -1)

{

    printf("Element is not present in the given array.");

}

else

{

    printf("Element is present at index: %d", result);

}
```

.

```
    return 0;

}
```

Features of Binary Search

a) It is great to search through large sorted arrays.
b) It has a time complexity of O(log n) which is a very good time complexity. We will discuss this in details in the.
c) It has a simple implementation.

Scope rules in C

- Scope of a variable is the visibility of that variable within the program or within function or block.
- C allows us to declare variables anywhere in the program. Unlike other programming language we need not declare them at the beginning of the program.
- Because of this feature, developer need not know all the variables that are required for the program.
- Consider a program to find the sum of two numbers. We can write this program in different ways: using single main program, by declaring the variables at the point of access, by using function within the program etc.

//Method 1

•

```
#include<stdio.h>
void main(){
	int intNum1, intNum2;
	int intResult;
	intNum1 = 50;
	intNum2 = 130;
	intResult = intNum1 + intNum2;
	printf("Sum of two number is : %d", intResult);
}
```

- In this method, all the variables that are accessed in the program are declared at the beginning of the main function. This is the traditional way of declaring the variables.
- These variables are available to access to any expression or statements throughout the main function. These variables cannot be accessed by any other function or block in the program or other program.
- The variables declared within the function or any block is called local variable to that function or block. That means scope of the local variables are limited to the function or block in which it is being declared and exists till the end of the function or block where it is declared.
- Whenever a local variable is declared, it is not automatically initialized. We need to explicitly assign value to it.

.

//Method 2

```
#include<stdio.h>
void main(){
        int intNum1, intNum2;
        intNum1 = 50;
        intNum2 = 130;
        int intResult = intNum1 + intNum2;
        printf("Sum of two number is : %d", intResult);
}
```

- In this method, result variable is declared when sum is calculated in the main program. Hence intResult comes into existence after it is being declared.
- If we try to access intResult before it is being declared, then the program will throw an error saying that intResult is not declared.
- Once it is declared, it can be accessed till the end of the main function. That means scope of the variable exists till the end of the block it is being declared.
- At the same time, intNum1 and intNum2 are declared at the beginning of the main function and can be accessed throughout the program.

//Method 3

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

•

```
int intResult; // Global variable
void main(){
        int intNum1, intNum2;
        intNum1 = 50;
        intNum2 = 130;
        intResult = intNum1 + intNum2;
        printf("Sum of two number is : %d", intResult);
}
```

- Here the variable intResult is declared outside the main function. It is not present in any other function or block.
- Hence it is called as **global variable**. Since this variable is not inside any block or function, it can be accessed by any function, block or expression.
- Hence the scope of global variable is not limited to any function or block; but it can be accessed by any function or block within the program. Global variables are initialized to the initial value defined for its datatype.

- Below is another method of adding two numbers. Here a separate function is written to add and display the result.

-

- Two local variables of main function – intNum1 and intNum2 are passed to the function. Since it is local variable, it cannot be accessed by any other functions in the program. Since it passed to the function, these variables can be accessed by the function AddNum ().

- But here we need to note the difference between the variables in the main function and AddNum function. The variables in the main function are local to it and can be accessed by it alone.

- These local parameters are passed to AddNum function, and are local to this function now. These parameters of function are called as formal parameter of a function.

- It can have same name as variables passed from the calling function or different. Suppose we have same name for local variable of main function and formal parameter.

- In this case, compiler considers both of them as different variables even though they have same name and value. Here memory address of variables at main function and AddNum function are different. This type of passing variable is called pass by value.
- Below we have given different names in both the functions. Even though they are having same value and same variable is passed to the function, both of them cannot be accessed in other functions.

•

- For example, if we try to access intNum1 and intNum2 in AddNum function, it will throw an error.
- Similarly, intResult is a local variable of AddNum and can be accesses only within it

//Method 4

```
#include<stdio.h>
void AddNum(int a, int b);
void main(){
        int intNum1, intNum2;
        intNum1 = 50;
        intNum2 = 130;
        AddNum(intNum1, intNum2);
//      printf("Sum of two number is : %d", intResult);
}

void AddNum(int a, int b){
        int intResult;
        intResult = a + b;
        printf("Sum of two number is : %d", intResult);
}
```

-

- Suppose we declare intResult as global variable. Then we can access it in either main function or in AddNum function. It will contain the value based on the last expression that is being evaluated.
- Consider the below program where intResult is declared as global variable. It can now be accessed from any function. It contains the value depending on the expression being evaluated in that function.

//Method 5

```
#include<stdio.h>
void AddNum(int a, int b);
int intResult;
void main(){
        int intNum1=100, intNum2 =200;
        AddNum(intNum1, intNum2);
intNum1 = 50;
intNum2 = 130;
intResult = intNum1 + intNum2;
printf("Sum of two number is : %d\n", intResult);
}

void AddNum(int a, int b){
```

•

```
    intResult = a + b;

    printf("Sum of two number is : %d\n", intResult);

}
```

- From all these examples, we understood that local variables are accessed only within the function or block it is being declared, whereas global variables are accessed throughout the program.
- Consider a block of code below:

//Method 6

```
    #include<stdio.h>

    void main(){

    int intArr[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

    for (int index =0, index<10, index++)

            printf("%d\t", intArr[index]);

    //printf("\nIndex = %d\n", index); // this row will throw an error

    }
```

- Here index variable is declared inside the 'for' loop. Even though index is inside the main function, it is declared inside a block in the main function.
- That means even though index is a local variable, for loop is considered as block and variables declared within it is local to it.

-

- Hence we can access index inside the for loop (need not be single line – it can have more than one line), but it cannot be accessed outside the loop – outside the block.
- The scope or life span of the variable index expires as soon as for loop ends – end of block.

Storage classes in C

- In C language, each variable has a storage class which decides the following things:
  a) **scope** i.e where the value of the variable would be available inside a program.
  b) **default initial value** i.e if we do not explicitly initialize that variable, what will be its default initial value.
  c) **lifetime** of that variable i.e for how long will that variable exist.

- The following storage classes are most oftenly used in C programming,

  Automatic variables

  External variables

•

Static variables

Register variables

Automatic variables: auto

Scope: Variable defined with auto storage class are local to the function block inside which they are defined.

Default Initial Value: Any random value i.e garbage value.

Lifetime: Till the end of the function/method block where the variable is defined.

- A variable declared inside a function without any storage class specification, is by default an automatic variable.
- They are created when a function is called and are destroyed automatically when the function's execution is completed.
- Automatic variables can also be called local variables because they are local to a function. By default they are assigned garbage value by the compiler.

```
#include<stdio.h>
void main()
{
        int detail;
        // or
        auto int details;    //Both are same
```

•

```
}
```

External or Global variable

Scope: Global i.e everywhere in the program. These variables are not bound by any function, they are available everywhere.

Default initial value: 0(zero).

Lifetime: Till the program doesn't finish its execution, you can access global variables.

- A variable that is declared outside any function is a Global Variable. Global variables remain available throughout the program execution.
- By default, initial value of the Global variable is 0(zero). One important thing to remember about global variable is that their values can be changed by any function in the program.

```
#include<stdio.h>
int number;     // global variable


void main()
{
        number = 10;
        printf("I am in main function. My value is %d\n", number);
        fun1();     //function calling, discussed in next topic
```

•

```
          fun2();     //function calling, discussed in next topic
     }

     /* This is function 1 */
     fun1()
     {
          number = 20;
          printf("I am in function fun1. My value is %d", number);
     }
     /* This is function 1 */
     fun2()
     {
          printf("\nI am in function fun2. My value is %d", number);
     }
```

OUTPUT:

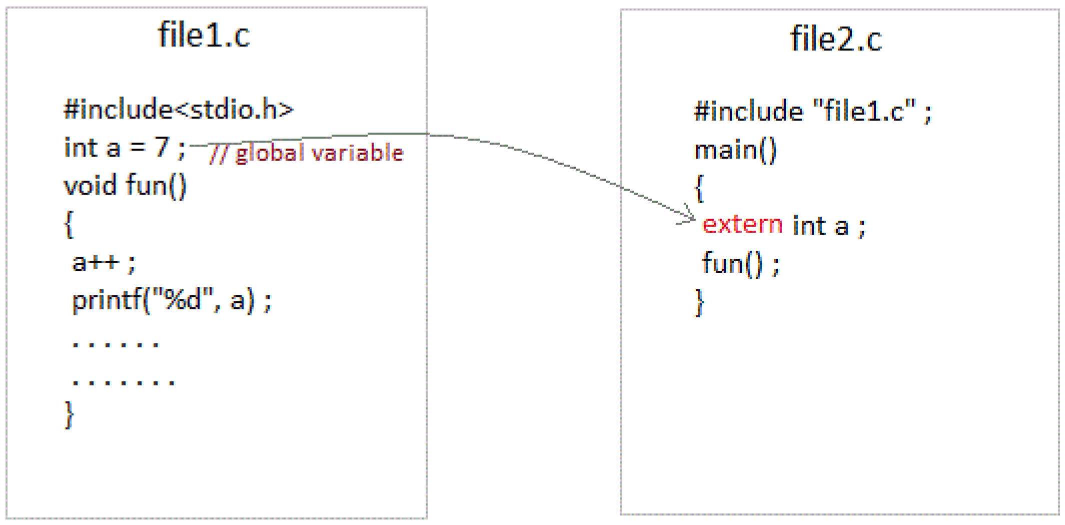
```
I am in function main. My value is 10
I am in function fun1. My value is 20
I am in function fun2. My value is 20
```

•

- Here the global variable number is available to all three functions and thus, if one function changes the value of the variable, it gets changed in every function.

extern keyword

- The extern keyword is used with a variable to inform the compiler that this variable is declared somewhere else.
- The extern declaration does not allocate storage for variables.



file1.c

```
#include<stdio.h>
int a = 7 ;  // global variable
void fun()
{
 a++ ;
 printf("%d", a) ;
 . . . . . .
 . . . . . . .
}
```

file2.c

```
#include "file1.c" ;
main()
{
 extern int a ;
 fun() ;
}
```

global variable from one file can be used in other using **extern** keyword.

- Problem when extern is not used

  int main()

  {

•

```
                a = 10;     //Error: cannot find definition of variable 'a'
                printf("%d", a);
        }
```

Example using extern in same file

```
int main()
{
    extern int x;   //informs the compiler that it is defined somewhere else
    x = 10;
    printf("%d", x);
}
int x;      //Global variable x
```

Static variables

Scope: Local to the block in which the variable is defined

Default initial value: 0(Zero).

Lifetime: Till the whole program doesn't finish its execution.

- A static variable tells the compiler to persist/save the variable until the end of program.
- Instead of creating and destroying a variable every time when it comes into and goes out of scope, static variable is initialized only once and remains into existence till the end of the program.

•

- A static variable can either be internal or external depending upon the place of declaration.
- Scope of internal static variable remains inside the function in which it is defined. External static variables remain restricted to scope of file in which they are declared.

They are assigned 0 (zero) as default value by the compiler.

```
#include<stdio.h>
void test();
int main()
{
   test();
   test();
   test();
}

void test()
{
   static int a = 0;       //a static variable
   a = a + 1;
   printf("%d\t",a);
}
```

•

1 2 3

Register variable

Scope: Local to the function in which it is declared.

Default initial value: Any random value i.e garbage value

Lifetime: Till the end of function/method block, in which the variable is defined.

- Register variables inform the compiler to store the variable in CPU register instead of memory. Register variables have faster accessibility than a normal variable.
- Generally, the frequently used variables are kept in registers. But only a few variables can be placed inside registers.
- One application of register storage class can be in using loops, where the variable gets used a number of times in the program, in a very short span of time.

NOTE: We can never get the address of such variables.

Syntax :

register int number;

Which storage class should be used and when

•

- To improve the speed of execution of the program and to carefully use the memory space occupied by the variables, following points should be kept in mind while using storage classes:
- We should use static storage class only when we want the value of the variable to remain same every time we call it using different function calls.
- We should use register storage class only for those variables that are used in our program very often. CPU registers are limited and thus should be used carefully.
- We should use external or global storage class only for those variables that are being used by almost all the functions in the program.
- If we do not have the purpose of any of the above mentioned storage classes, then we should use the automatic storage class

Bitwise operators

Bitwise operators perform manipulations of data at **bit level**. These operators also perform **shifting of bits** from right to left. Bitwise operators are not applied to float or double

Operator	Description
&	Bitwise AND
|	Bitwise OR

•

^	Bitwise exclusive OR
<<	left shift
>>	right shift

Now lets see truth table for bitwise &, | and ^

a	**B**	**a & b**	**a | b**	**a ^ b**
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

The bitwise **shift** operator, shifts the bit value. The left operand specifies the value to be shifted and the right operand specifies the number of positions that the bits in the value have to be shifted. Both operands have the same precedence.

.

MODULE 6

FILES IN C

File Input/Output in C

A file represents a sequence of bytes on the disk where a group of related data is stored. File is created for permanent storage of data.

In C language, we use a structure pointer of file type to declare a file.

FILE *fp;

C provides a number of functions that helps to perform basic file operations. Following are the functions,

Function	Description
fopen()	create a new file or open a existing file
fclose()	closes a file
getc()	reads a character from a file
putc()	writes a character to a file
fscanf()	reads a set of data from a file

•

fprintf()	writes a set of data to a file
getw()	reads a integer from a file
putw()	writes a integer to a file
fseek()	set the position to desire point
ftell()	gives current position in the file
rewind()	set the position to the begining point

Opening a File or Creating a File

The fopen() function is used to create a new file or to open an existing file.

General Syntax:

*fp = FILE *fopen(const char *filename, const char *mode);

Here, *fp is the FILE pointer (FILE *fp), which will hold the reference to the opened (or created) file.

filename is the name of the file to be opened and mode specifies the purpose of opening the file. Mode can be of following types,

mode	Description

•

r	opens a text file in reading mode
w	opens or create a text file in writing mode.
a	opens a text file in append mode
r+	opens a text file in both reading and writing mode
w+	opens a text file in both reading and writing mode
a+	opens a text file in both reading and writing mode
rb	opens a binary file in reading mode
wb	opens or create a binary file in writing mode
ab	opens a binary file in append mode
rb+	opens a binary file in both reading and writing mode
wb+	opens a binary file in both reading and writing mode
ab+	opens a binary file in both reading and writing mode

Closing a File

•

The fclose() function is used to close an already opened file.

General Syntax :

```
int fclose( FILE *fp);
```

Here fclose() function closes the file and returns zero on success, or EOF if there is an error in closing the file. This EOF is a constant defined in the header file stdio.h.

Input/Output operation on File

In the above table we have discussed about various file I/O functions to perform reading and writing on file. getc() and putc() are the simplest functions which can be used to read and write individual characters to a file.

```
#include<stdio.h>
int main()
{
    FILE *fp;
    char ch;
    fp = fopen("one.txt", "w");
    printf("Enter data...");
    while( (ch = getchar()) != EOF) {
        putc(ch, fp);
    }
```

•

```
    fclose(fp);

    fp = fopen("one.txt", "r");

    while( (ch = getc(fp)! = EOF)

    printf("%c",ch);

    // closing the file pointer

    fclose(fp);

    return 0;

}
```

Reading and Writing to File using fprintf() and fscanf()

```
#include<stdio.h>

struct emp

{

    char name[10];

    int age;

};

void main()

{

    struct emp e;

    FILE *p,*q;
```

•

```
    p = fopen("one.txt", "a");
    q = fopen("one.txt", "r");
    printf("Enter Name and Age:");
    scanf("%s %d", e.name, &e.age);
    fprintf(p,"%s %d", e.name, e.age);
    fclose(p);
    do
    {
        fscanf(q,"%s %d", e.name, e.age);
        printf("%s %d", e.name, e.age);
    }
    while(!feof(q));
}
```

In this program, we have created two FILE pointers and both are refering to the same file but in different modes.

fprintf() function directly writes into the file, while fscanf() reads from the file, which can then be printed on the console using standard printf() function.

.

Difference between Append and Write Mode

Write (w) mode and Append (a) mode, while opening a file are almost the same. Both are used to write in a file. In both the modes, new file is created if it doesn't exists already.

The only difference they have is, when you open a file in the write mode, the file is reset, resulting in deletion of any data already present in the file. While the append mode is used to append or add data to the existing data of file(if any). Hence, when you open a file in Append(a) mode, the cursor is positioned at the end of the present data in the file.

Formatted and Unformatted Input/Output

- Unformatted Input/Output is the most basic form of input/output. Unformatted input/output transfers the internal binary representation of the data directly between memory and the file.
- Formatted output converts the internal binary representation of the data to ASCII characters which are written to the output file.
- Formatted input reads characters from the input file and converts them to internal form. Formatted I/O can be either "Free" format or "Explicit" format.

Advantages and Disadvantages of Unformatted I/O

- Unformatted input/output is the simplest and most efficient form of input/output.

.

- It is usually the most compact way to store data.
- Unformatted input/output is the least portable form of input/output. Unformatted data files can only be moved easily to and from computers that share the same internal data representation.
- It should be noted that XDR (eXternal Data Representation) files, can be used to produce portable binary data.
- Unformatted input/output is not directly human readable, so you cannot type it out on a terminal screen or edit it with a text editor.

Advantages and Disadvantages of Formatted I/O

- Formatted input/output is very portable. It is a simple process to move formatted data files to various computers, even computers running different operating systems, as long as they all use the ASCII character set
- Formatted files are human readable and can be typed to the terminal screen or edited with a text editor
- However, formatted input/output is more computationally expensive than unformatted input/output because of the need to convert between internal binary data and ASCII text.
- Formatted data requires more space than unformatted to represent the same information. Inaccuracies can result when converting data between text and the internal representation.

Free Format I/O

.

With free format input/output, IDL uses default rules to format the data.

Advantages and Disadvantages of Free Format I/O

- The user is free of the chore of deciding how the data should be formatted. Free format is extremely simple and easy to use.
- It provides the ability to handle the majority of formatted input/output needs with a minimum of effort.
- However, the default formats used are not always exactly what is required. In this case, explicit formatting is necessary.

Explicit Format I/O

Explicit format I/O allows you to specify the exact format for input/output.

Advantages and Disadvantages of Explicit I/O

- Explicit formatting allows a great deal of flexibility in specifying exactly how data will be formatted.
- Formats are specified using a syntax that is similar to that used in FORTRAN format statements.
- Scientists and engineers already familiar with FORTRAN will find IDL formats easy to write.
- Commonly used FORTRAN format codes are supported. In addition, IDL formats have been extended to provide many of the capabilities found in

•

the scanf () and printf () functions commonly found in the C language runtime library.

- However, there are some disadvantages to using Explicit I/O. Using explicitly specified formats requires the user to specify more detail-they are, therefore, more complicated to use than free format.
- The type of input/output to use in a given situation is usually determined by considering the advantages and disadvantages of each method as they relate to the problem to be solved.
- Also, when transferring data to or from other programs or systems, the type of input/output is determined by the application.
- The following suggestions are intended to give a rough idea of the issues involved, though there are always exceptions:

a) Images and large data sets are usually stored and manipulated using unformatted input/output in order to minimize processing overhead. The IDL ASSOC function is often the natural way to access such data.

b) Data that need to be human readable should be written using formatted input/output.

c) Data that need to be portable should be written using formatted input/output. Another option is to use unformatted XDR files by specifying the XDR keyword with the OPEN procedures. This is especially important if moving between computers with markedly different internal binary data formats.

d) Free format input/output is easier to use than explicitly formatted input/output and about as easy as unformatted input/output, so it is often a good choice for small files where there is no strong reason to prefer one method over another.

.

e) Special well-known complex file formats are usually supported directly with special IDL routines (e.g. READ_JPEG for JPEG images)

Command Line Argument in C

- Command line argument is a parameter supplied to the program when it is invoked.
- Command line argument is an important concept in C programming.
- It is mostly used when you need to control your program from outside.
- Command line arguments are passed to the main() method.
- Syntax:

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

- Here argc counts the number of arguments on the command line and argv[ ] is a pointer array which holds pointers of type char which points to the arguments passed to the program.

Example for Command Line Argument

```
#include <stdio.h>
#include <conio.h>
int main(int argc, char *argv[])
```

•

```
{
   int i;
   if( argc >= 2 )
   {
      printf("The arguments supplied are:\n");
      for(i = 1; i < argc; i++)
      {
         printf("%s\t", argv[i]);
      }
   }
   else
   {
      printf("argument list is empty.\n");
   }
   return 0;
}
```

- Remember that argv[0] holds the name of the program and argv[1] points to the first command line argument and argv[n] gives the last argument.
- If no argument is supplied, argc will be 1

•

CONTENT BEYOND SYLLABUS

3D Array

You can declare a three-dimensional (3d) array. For example,

```
float y[2][4][3];
```

Here, the array y can hold 24 elements.

Initialization of a 3d array

You can initialize a three-dimensional array in a similar way like a two-dimensional array. Here's an example,

```
int test[2][3][4] = {
    {{3, 4, 2, 3}, {0, -3, 9, 11}, {23, 12, 23, 2}},
    {{13, 4, 56, 3}, {5, 9, 3, 5}, {3, 1, 4, 9}}};
```

•

ENUMERATION (ENUM) IN C

Enumeration (or enum) is a user defined data type in C. It is mainly used to assign names to integral constants, the names make a program easy to read and maintain.

The keyword ‘enum’ is used to declare new enumeration types in C and C++. Following is an example of enum declaration.

```
// The name of enumeration is "flag" and the constant

// are the values of the flag. By default, the values

// of the constants are as follows:

// constant1 = 0, constant2 = 1, constant3 = 2 and

// so on.

enum flag{constant1, constant2, constant3, ....... };
```

Variables of type enum can also be defined. They can be defined in two ways:

```
// In both of the below cases, "day" is
```

•

```
// defined as the variable of type week.



enum week{Mon, Tue, Wed};

enum week day;

// Or

enum week{Mon, Tue, Wed}day;
```