



# LEARN C PROGRAMMING



**CH SRILAXMI**

## Author Profile:

**Prof. Srilakshmi Cherukuri working as an Assistant Professor & HoD in the CSE(AI &ML)Department at Narsimha Reddy Engineering College, Hyderabad. She secured a Master of Technology in CSE. She is pursuing a Ph.D., in GITAM University, Hyderabad, India. She is in the field of teaching profession for more than 18 years. She has presented more than 25 papers in National and International Journals, Conference and Symposiums. Her main area of interest includes Deep Learning and Image Processing. Throughout her career, Ch Srilakahmi has been passionate about teaching and sharing her knowledge with others. She has conducted numerous workshops and seminars on programming languages, with a particular focus on C programming. Ch Srilakshmi's deep understanding of C programming stems from her hands-on experience in developing software solutions for diverse applications, including embedded systems, operating systems, and game development. Her practical approach to teaching, combined with real-world examples, makes complex concepts easy to understand for beginners. "Learn C Programming" is Ch Srilakshmi's latest endeavor to make programming accessible to enthusiasts and aspiring developers. In the book, She distills her years of experience into a comprehensive guide that covers everything from the basics of C syntax to advanced programming techniques. Ch Srilakshmi's commitment to helping others succeed in programming is evident in her's clear explanations, step-by-step instructions, and practical exercises designed to reinforce learning. Whether you're a student, a professional looking to expand your skill set, or simply someone curious about the world of programming, "Learn C Programming" is the perfect resource to kickstart your journey. In addition to writing and teaching, Ch Srilakshmi enjoys hiking, playing the guitar, and spending time with her family in her spare time.**

# PREFACE

Welcome to "Learn C Programming"!

C programming language holds a special place in the world of computer science and software development. Its simplicity, efficiency, and versatility have made it a cornerstone of modern computing. Whether you're an aspiring programmer taking your first steps into the vast world of coding or a seasoned developer looking to deepen your understanding, this book is crafted to be your guide.

In this comprehensive guide, we embark on a journey through the fundamental concepts and principles of C programming. From the basics of syntax and control structures to advanced topics such as memory management and data structures, each chapter is meticulously designed to build upon the previous one, providing you with a solid foundation of knowledge.

But this book is more than just a compilation of code snippets and theoretical explanations. It's a companion on your learning path, offering hands-on exercises, real-world examples, and practical insights to reinforce your understanding and sharpen your skills. Whether you're exploring the intricacies of pointers or unraveling the mysteries of function pointers, each concept is presented in a clear, concise manner, empowering you to grasp even the most complex concepts with ease.

As you progress through these pages, remember that mastery of C programming is not merely about memorizing syntax or regurgitating algorithms. It's about cultivating a mindset of problem-solving, creativity, and continuous learning. It's about embracing the challenges that come with mastering a powerful tool like C and leveraging its capabilities to create innovative solutions to real-world problems.

So, whether you're embarking on your first "Hello, World!" program or delving into the depths of multi-threaded programming, let this book be your trusted companion on your journey to becoming a proficient C programmer. Let's dive in and unlock the boundless possibilities that await you in the world of C programming.

Happy coding!

Ch Srilaxmi

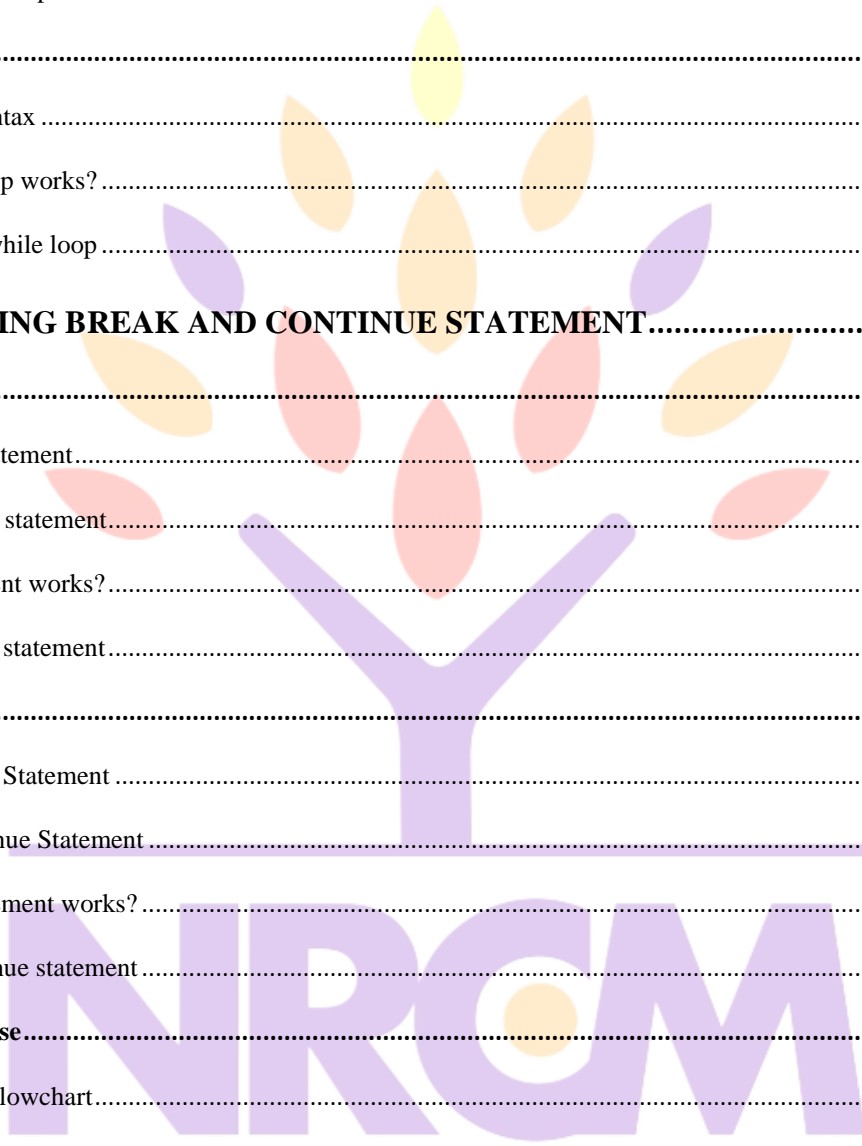
<b>LEARN C PROGRAMMING .....</b>	<b>14</b>
<b>C PROGRAMMING KEYWORDS AND IDENTIFIERS .....</b>	<b>21</b>
<b>Character set .....</b>	<b>21</b>
Alphabets .....	22
Digits .....	22
Special Characters .....	22
<b>C Keywords.....</b>	<b>22</b>
<b>C Identifiers .....</b>	<b>23</b>
Rules for writing an identifier.....	23
<b>Variables .....</b>	<b>23</b>
Rules for naming a variable in C .....	24
<b>Constants/Literals: A constant is a value or an identifier whose value cannot be altered in a program. For example: 1, 2.5, "C programming is easy", etc.....</b>	<b>24</b>
1. Integer constants .....	24
2. Floating-point constants.....	25
3. Character constants .....	25
4. Escape Sequences .....	25
5. String constants.....	26
6. Enumeration constants.....	26
<b>C PROGRAMMING INPUT OUTPUT (I/O): PRINTF() AND SCANF() .....</b>	<b>29</b>
<b>Example #1: C Output .....</b>	<b>29</b>
<b>Example #2: C Integer Output .....</b>	<b>29</b>
<b>#include &lt;stdio.h&gt;int main(){ .....</b>	<b>29</b>
<b>Example #3: C Integer Input/Output .....</b>	<b>29</b>
<b>Example #3: C Floats Input/Output .....</b>	<b>30</b>
<b>Example #4: C Character I/O .....</b>	<b>30</b>
Little bit on ASCII code .....	30
<b>Example #5: C ASCII Code .....</b>	<b>30</b>
<b>Example #6: C ASCII Code .....</b>	<b>31</b>
<b>More on Input/Output of floats and Integers .....</b>	<b>31</b>

Example #7: I/O of Floats and Integers .....	31
<b>C PROGRAMMING INPUT OUTPUT (I/O): PRINTF() AND SCANF() .....</b>	<b>32</b>
<b>Example #1: C Output .....</b>	<b>32</b>
<b>Example #2: C Integer Output .....</b>	<b>33</b>
<b>Example #3: C Integer Input/Output .....</b>	<b>33</b>
<b>Example #3: C Floats Input/Output .....</b>	<b>33</b>
<b>Example #4: C Character I/O .....</b>	<b>33</b>
Little bit on ASCII code .....	34
<b>Example #5: C ASCII Code .....</b>	<b>34</b>
<b>Example #6: C ASCII Code .....</b>	<b>34</b>
<b>More on Input/Output of floats and Integers .....</b>	<b>35</b>
Example #7: I/O of Floats and Integers .....	35
<b>C PREPROCESSOR AND MACROS .....</b>	<b>35</b>
<b>Including Header Files .....</b>	<b>36</b>
<b>Macros using #define .....</b>	<b>36</b>
Example 1: Using #define preprocessor .....	36
Example 2: Using #define preprocessor .....	37
<b>Conditional Compilation .....</b>	<b>38</b>
Uses of Conditional .....	38
How to use conditional? .....	38
<b>Predefined Macros .....</b>	<b>39</b>
Example #3: predefined Macros .....	40
<b>C STANDARD LIBRARY FUNCTIONS .....</b>	<b>40</b>
<b>Advantages of using C library functions .....</b>	<b>41</b>
1. They work .....	41
2. The functions are optimized for performance .....	41
3. It saves considerable development time .....	41
3. The functions are portable .....	41
<b>Use Of Library Function To Find Square root .....</b>	<b>41</b>
<b>C Library Functions Under Different Header File .....</b>	<b>42</b>



Example #1: Arithmetic Operators .....	43
<b>Increment and decrement operators .....</b>	<b>44</b>
Example #2: Increment and Decrement Operators .....	45
<b>C Assignment Operators.....</b>	<b>45</b>
<b>Example #3: Assignment Operators .....</b>	<b>46</b>
C Relational Operators .....	47
Example #4: Relational Operators .....	47
C Logical Operators.....	48
Example #5: Logical Operators .....	49
Bitwise Operators .....	50
<b>Other Operators.....</b>	<b>50</b>
Comma Operator .....	50
The sizeof operator .....	50
Example #6: sizeof Operator .....	51
C-Ternary Operator (?: ).....	51
<b>C IF, IF...ELSE AND NESTED IF...ELSE STATEMENT.....</b>	<b>52</b>
<b>C if statement .....</b>	<b>52</b>
Flowchart of if statement .....	53
Example #1: C if statement.....	53
<b>C if...else statement .....</b>	<b>54</b>
Syntax of if...else .....	54
Flowchart of if...else statement.....	54
Example #2: C if...else statement .....	55
<b>Nested if...else statement (if...elseif...else Statement) .....</b>	<b>56</b>
Example #3: C Nested if...else statement .....	56
<b>C PROGRAMMING FOR LOOP.....</b>	<b>57</b>
<b>for Loop.....</b>	<b>58</b>
How for loop works? .....	58
for loop Flowchart .....	58
Example: for loop .....	58

<b>C PROGRAMMING WHILE AND DO...WHILE LOOP .....</b>	<b>59</b>
<b>while loop .....</b>	<b>60</b>
How while loop works?.....	60
Flowchart of while loop.....	60
Example #1: while loop.....	60
<b>do...while loop.....</b>	<b>61</b>
do...while loop Syntax .....	61
How do...while loop works?.....	62
Example #2: do...while loop.....	62
<b>C PROGRAMMING BREAK AND CONTINUE STATEMENT.....</b>	<b>63</b>
<b>break Statement .....</b>	<b>63</b>
Syntax of break statement.....	63
Flowchart of break statement.....	63
How break statement works?.....	64
Example #1: break statement.....	64
<b>continue Statement.....</b>	<b>65</b>
Syntax of continue Statement .....	65
Flowchart of continue Statement .....	65
How continue statement works?.....	66
Example #2: continue statement.....	66
<b>Syntax of switch...case.....</b>	<b>68</b>
switch Statement Flowchart.....	69
Example: switch Statement// Program to create a simple calculator// Performs addition, subtraction, multiplication or division depending the input from user .....	69
<b>C GOTO STATEMENT .....</b>	<b>71</b>
Syntax of goto statement .....	71
Example: goto Statement.....	72
Reasons to avoid goto statement.....	73
<b>C PROGRAMMING FUNCTIONS.....</b>	<b>73</b>
<b>Types of functions in C programming.....</b>	<b>73</b>



your roots to success...

Standard library functions.....	74
User-defined functions.....	74
<b>How user-defined function works? .....</b>	<b>74</b>
Advantages of user-defined function .....	76
<b>C PROGRAMMING USER-DEFINED FUNCTIONS .....</b>	<b>77</b>
<b>Example: User-defined function .....</b>	<b>77</b>
<b>Function prototype.....</b>	<b>78</b>
Syntax of function prototype .....	78
<b>Calling a function.....</b>	<b>78</b>
Syntax of function call.....	78
<b>Function definition.....</b>	<b>78</b>
<b>Passing arguments to a function .....</b>	<b>79</b>
<b>Return Statement.....</b>	<b>79</b>
Syntax of return statement .....	80
<b>TYPES OF USER-DEFINED FUNCTIONS IN C PROGRAMMING .....</b>	<b>80</b>
<b>Example #1: No arguments passed and no return Value .....</b>	<b>81</b>
<b>Example #2: No arguments passed but a return value .....</b>	<b>82</b>
<b>Example #3: Argument passed but no return value .....</b>	<b>83</b>
<b>Example #4: Argument passed and a return value .....</b>	<b>84</b>
<b>Which approach is better? .....</b>	<b>85</b>
<b>C PROGRAMMING RECURSION.....</b>	<b>85</b>
How recursion works? .....	85
Example: Sum of Natural Numbers Using Recursion .....	86
Advantages and Disadvantages of Recursion .....	88
<b>SCOPE AND LIFETIME OF A VARIABLE.....</b>	<b>88</b>
<b>Local Variable.....</b>	<b>89</b>
<b>Global Variable.....</b>	<b>89</b>
Example #1: External Variable.....	89
<b>Register Variable .....</b>	<b>89</b>

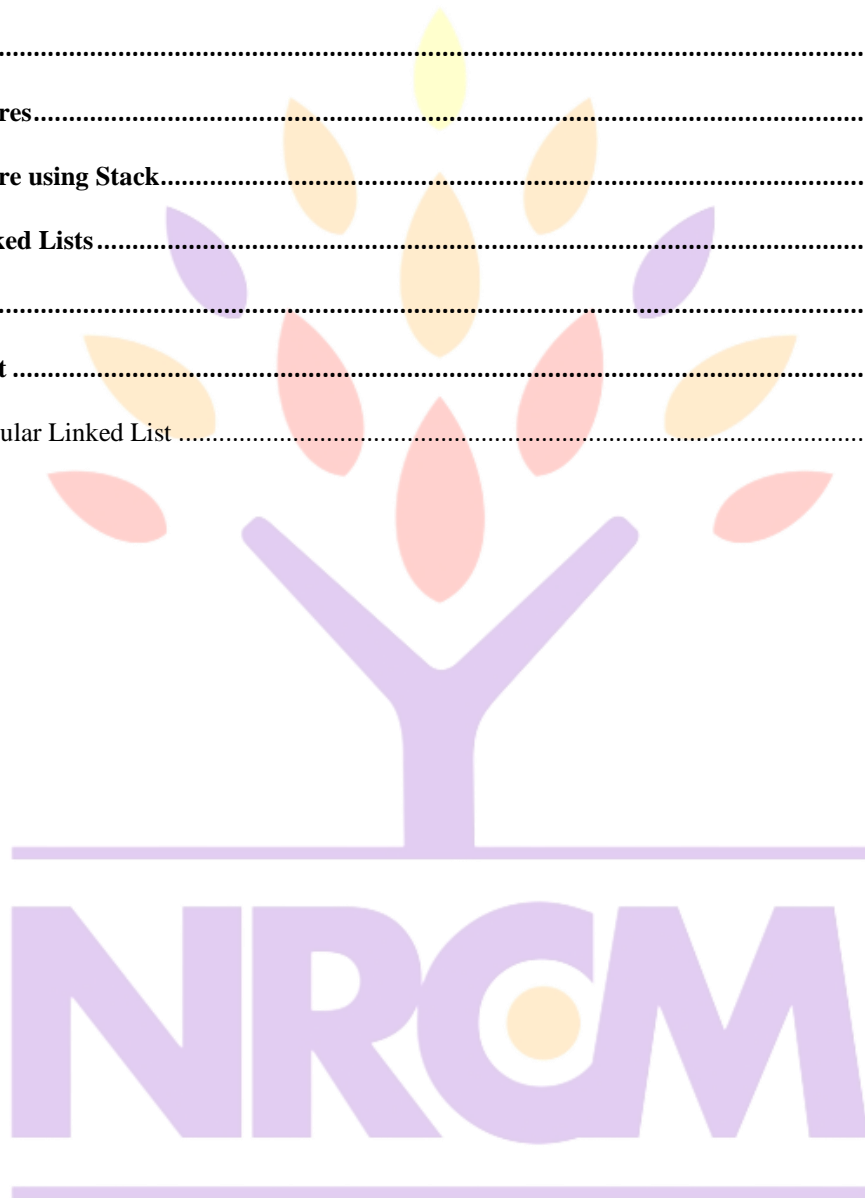


<b>Static Variable.....</b>	<b>90</b>
Example #2: Static Variable .....	90
<b>C PROGRAMMING ARRAYS .....</b>	<b>90</b>
<b>How to declare an array in C? .....</b>	<b>91</b>
<b>Elements of an Array and How to access them?.....</b>	<b>91</b>
How to initialize an array in C programming? .....	91
How to insert and print array elements? .....	92
<b>Example: C Arrays .....</b>	<b>92</b>
Important thing to remember when working with C arrays.....	93
<b>C PROGRAMMING MULTIDIMENSIONAL ARRAYS .....</b>	<b>93</b>
<b>How to initialize a multidimensional array?.....</b>	<b>94</b>
Initialization of a two dimensional array .....	94
Initialization of a three dimensional array.....	94
Example #1: Two Dimensional Array to store and display values .....	95
Example #2: Sum of two matrices using Two dimensional arrays .....	97
Example 3: Three Dimensional Array .....	98
<b>HOW TO PASS ARRAYS TO A FUNCTION IN C PROGRAMMING? .....</b>	<b>100</b>
<b>Passing One-dimensional Array In Function .....</b>	<b>100</b>
Passing an entire one-dimensional array to a function .....	101
<b>Passing Multi-dimensional Arrays to Function.....</b>	<b>101</b>
#Example: Pass two-dimensional arrays to a function .....	101
<b>C PROGRAMMING POINTERS AND ARRAYS.....</b>	<b>103</b>
<b>Relation between Arrays and Pointers .....</b>	<b>103</b>
Example: Program to find the sum of six numbers with arrays and pointers .....	104
<b>C CALL BY REFERENCE: USING POINTERS [WITH EXAMPLES] .....</b>	<b>105</b>
Example of Pointer And Functions.....	105
<b>C DYNAMIC MEMORY ALLOCATION.....</b>	<b>106</b>
<b>C malloc().....</b>	<b>107</b>
Syntax of malloc() .....	107

<b>C calloc()</b> .....	<b>107</b>
Syntax of calloc().....	107
<b>C free()</b> .....	<b>108</b>
syntax of free().....	108
Example #1: Using C malloc() and free().....	108
Example #2: Using C calloc() and free() .....	109
<b>C realloc()</b> .....	<b>110</b>
Syntax of realloc() .....	110
Example #3: Using realloc().....	110
<b>C PROGRAMMING STRUCTURE</b> .....	<b>110</b>
<b>Structure Definition in C</b> .....	<b>111</b>
Syntax of structure.....	111
<b>Structure variable declaration</b> .....	<b>111</b>
<b>Accessing members of a structure</b> .....	<b>112</b>
Example of structure.....	112
Keyword typedef while using structure .....	114
<b>Structures within structures</b> .....	<b>115</b>
<b>Passing structures to a function</b> .....	<b>115</b>
<b>C PROGRAMMING STRUCTURE AND POINTER</b> .....	<b>115</b>
<b>Accessing structure's member through pointer</b> .....	<b>116</b>
1. Referencing pointer to another address to access the memory_ Consider an example to access structure's member through pointer.....	116
2. Accessing structure member through pointer using dynamic memory allocation .....	117
<b>HOW TO PASS STRUCTURE TO A FUNCTION IN C PROGRAMMING?</b> .....	<b>118</b>
Passing structure by value .....	119
Passing structure by reference .....	120
<b>C PROGRAMMING UNIONS</b> .....	<b>121</b>
<b>How to create union variables?</b> .....	<b>121</b>
Accessing members of a union .....	122
<b>Difference between union and structure</b> .....	<b>122</b>

More memory is allocated to structures than union .....	123
Only one union member can be accessed at a time.....	124
Passing Union To a Function.....	125
<b>C PROGRAMMING FILES I/O .....</b>	<b>125</b>
<b>Why files are needed? .....</b>	<b>125</b>
<b>Types of Files.....</b>	<b>125</b>
1. Text files .....	125
2. Binary files .....	126
<b>File Operations.....</b>	<b>126</b>
<b>Working with files.....</b>	<b>126</b>
<b>Opening a file - for creation and edit.....</b>	<b>126</b>
<b>Closing a File.....</b>	<b>127</b>
<b>Reading and writing to a text file.....</b>	<b>127</b>
Writing to a text file.....	128
Reading from a text file .....	128
<b>Reading and writing to a binary file .....</b>	<b>129</b>
Writing to a binary file .....	129
Reading from a binary file .....	131
<b>Getting data using fseek() .....</b>	<b>132</b>
Syntax of fseek().....	132
Example of fseek().....	132
<b>C PROGRAMMING ENUMERATION .....</b>	<b>134</b>
<b>Enumerated Type Declaration.....</b>	<b>134</b>
Example: Enumeration Type .....	135
<b>Why enums are used in C programming? .....</b>	<b>135</b>
How to use enums for flags?.....	136
<b>Introduction to Data Structures .....</b>	<b>150</b>
<b>Time Complexity of Algorithms .....</b>	<b>152</b>
<b>Introduction to Sorting .....</b>	<b>154</b>
<b>Bubble Sorting .....</b>	<b>155</b>

Insertion Sorting .....	157
Quick Sort Algorithm .....	160
Merge Sort Algorithm.....	162
Heap Sort Algorithm.....	164
Searching Algorithms on Array.....	168
Stacks .....	170
Queue Data Structures.....	173
Queue Data Structure using Stack.....	177
Introduction to Linked Lists.....	179
Linear Linked List .....	180
Circular Linked List .....	185
Implementing Circular Linked List .....	186



your roots to success...

## Introduction to Programming

A program is a set of instructions that tell the computer to do various things; sometimes the instruction it has to perform depends on what happened when it performed a previous instruction. This section gives an overview of the two main ways in which you can give these instructions, or “commands” as they are usually called. One way uses an interpreter, the other a compiler. As human languages are too difficult for a computer to understand in an unambiguous way, commands are usually written in one or other languages specially designed for the purpose.

### Interpreters

With an interpreter, the language comes as an environment, where you type in commands at a prompt and the environment executes them for you. For more complicated programs, you can type the commands into a file and get the interpreter to load the file and execute the commands in it. If anything goes wrong, many interpreters will drop you into a debugger to help you track down the problem.

The advantage of this is that you can see the results of your commands immediately, and mistakes can be corrected readily. The biggest disadvantage comes when you want to share your programs with someone. They must have the same interpreter, or you must have some way of giving it to them, and they need to understand how to use it. Also users may not appreciate being thrown into a debugger if they press the wrong key! From a performance point of view, interpreters can use up a lot of memory, and generally do not generate code as efficiently as compilers.

In my opinion, interpreted languages are the best way to start if you have not done any programming before. This kind of environment is typically found with languages like Lisp, Smalltalk, Perl and Basic. It could also be argued that the UNIX® shell (sh, csh) is itself an interpreter, and many people do in fact write shell “scripts” to help with various “housekeeping” tasks on their machine. Indeed, part of the original UNIX® philosophy was to provide lots of small utility programs that could be linked together in shell scripts to perform useful tasks.

### Interpreters available with FreeBSD

Here is a list of interpreters that are available from the FreeBSD Ports Collection, with a brief discussion of some of the more popular interpreted languages.

Instructions on how to get and install applications from the Ports Collection can be found in the Ports section of the handbook.

### BASIC

Short for Beginner's All-purpose Symbolic Instruction Code. Developed in the 1950s for teaching University students to program and provided with every self-respecting personal computer in the 1980s, BASIC has been the first programming language for many programmers. It is also the foundation for Visual Basic. The Bywater Basic Interpreter can be found in the Ports Collection as lang/bwbasic and the Phil Cockroft's Basic Interpreter (formerly Rabbit Basic) is available as lang/pbasic.

### Lisp

A language that was developed in the late 1950s as an alternative to the “number-crunching” languages that were popular at the time. Instead of being based on numbers, Lisp is based on lists; in fact, the name is short for “List Processing”. It is very popular in AI (Artificial Intelligence) circles.

Lisp is an extremely powerful and sophisticated language, but can be rather large and unwieldy.

Various implementations of Lisp that can run on UNIX® systems are available in the Ports Collection for FreeBSD. GNU Common Lisp can be found as lang/gcl. CLISP by Bruno Haible and Michael Stoll is available as lang/clisp. For CMUCL, which includes a highly-optimizing compiler too, or simpler Lisp implementations like SLisp, which implements most of the Common Lisp constructs in a few hundred lines of C code, lang/cmucl and lang/slip are available respectively.

### **Perl**

Very popular with system administrators for writing scripts; also often used on World Wide Web servers for writing CGI scripts.

Perl is available in the Ports Collection as lang/perl5.16 for all FreeBSD releases.

### **Scheme**

A dialect of Lisp that is rather more compact and cleaner than Common Lisp. Popular in Universities as it is simple enough to teach to undergraduates as a first language, while it has a high enough level of abstraction to be used in research work.

Scheme is available from the Ports Collection as lang/elk for the Elk Scheme Interpreter. The MIT Scheme Interpreter can be found in lang/mit-scheme and the SCM Scheme Interpreter in lang/scm.

### **Icon**

Icon is a high-level language with extensive facilities for processing strings and structures. The version of Icon for FreeBSD can be found in the Ports Collection as lang/icon.

### **Logo**

Logo is a language that is easy to learn, and has been used as an introductory programming language in various courses. It is an excellent tool to work with when teaching programming to smaller age groups, as it makes creation of elaborate geometric shapes an easy task.

The latest version of Logo for FreeBSD is available from the Ports Collection in lang/logo.

### **Python**

Python is an Object-Oriented, interpreted language. Its advocates argue that it is one of the best languages to start programming with, since it is relatively easy to start with, but is not limited in comparison to other popular interpreted languages that are used for the development of large, complex applications (Perl and Tcl are two other languages that are popular for such tasks).

The latest version of Python is available from the Ports Collection in lang/python.

### **Ruby**



Ruby is an interpreter, pure object-oriented programming language. It has become widely popular because of its easy to understand syntax, flexibility when writing code, and the ability to easily develop and maintain large, complex programs.

Ruby is available from the Ports Collection as lang/ruby18.

### **Tcl and Tk**

Tcl is an embeddable, interpreted language, that has become widely used and became popular mostly because of its portability to many platforms. It can be used both for quickly writing small, prototype applications, or (when combined with Tk, a GUI toolkit) fully-fledged, featureful programs.

Various versions of Tcl are available as ports for FreeBSD. The latest version, Tcl 8.5, can be found in lang/tcl85.

### **Compilers**

Compilers are rather different. First of all, you write your code in a file (or files) using an editor. You then run the compiler and see if it accepts your program. If it did not compile, grit your teeth and go back to the editor; if it did compile and gave you a program, you can run it either at a shell command prompt or in a debugger to see if it works properly. [1]

Obviously, this is not quite as direct as using an interpreter. However it allows you to do a lot of things which are very difficult or even impossible with an interpreter, such as writing code which interacts closely with the operating system—or even writing your own operating system! It is also useful if you need to write very efficient code, as the compiler can take its time and optimize the code, which would not be acceptable in an interpreter. Moreover, distributing a program written for a compiler is usually more straightforward than one written for an interpreter—you can just give them a copy of the executable, assuming they have the same operating system as you.

As the edit-compile-run-debug cycle is rather tedious when using separate programs, many commercial compiler makers have produced Integrated Development Environments (IDEs for short). FreeBSD does not include an IDE in the base system, but devel/kdevelop is available in the Ports Collection and many use Emacs for this purpose. Using Emacs as an IDE is discussed in Section 2.7, “Using Emacs as a Development Environment”.

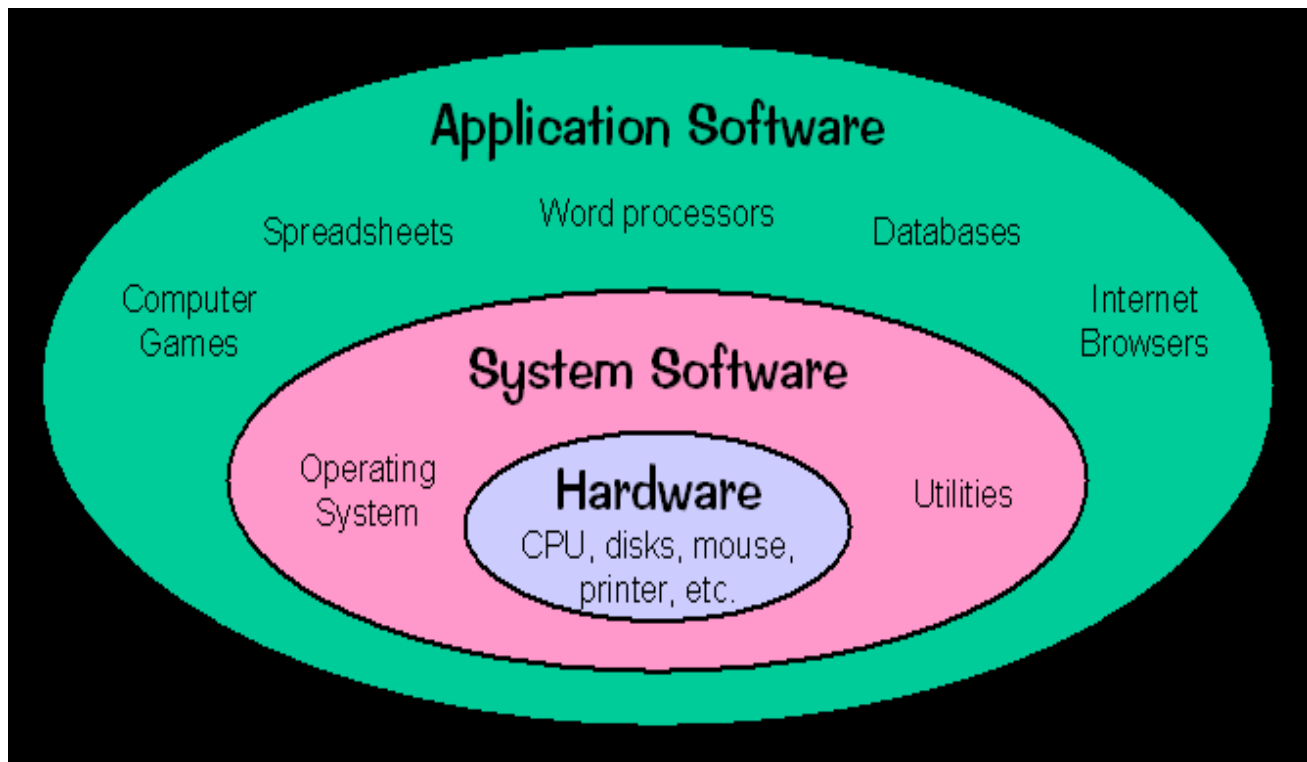
### **Software:**

Organized information in the form of operating systems, utilities, programs, and applications that enable computers to work.

Software consists of carefully-organized instructions and code written by programmers in any of various special computer languages. Software is divided commonly into two main categories:

(1) **System software:** controls the basic (and invisible to the user) functions of a computer and comes usually preinstalled with the machine. See also BIOS and Operating System.

(2) **Application software:** handles multitudes of common and specialized tasks a user wants to perform, such as accounting, communicating, data processing, word processing.



**Compiler:** Translating from source code to machine code executing directly.

**Interpreter:** Translating from source code to machine code executing it line-by-line.

**Assembler:** Translating from Assembly language to machine language.

conversation of source to machine code

Binary Evaluate	$2^4$	$2^3$	$2^2$	$2^1$	$2^0$		Value	Decimal Number
					0	>	0	0
				1		>	1	1
			1	1	0	>	$4+2+0$	6
		1	0	1	0	>	$8+0+2+0$	10
	1	0	1	1	0	>	$16+0+4+2+0$	22
	1	1	0	0	1	>	$16+8+0+0+1$	25
	1	1	1	1	1	>	$16+8+4+2+1$	31

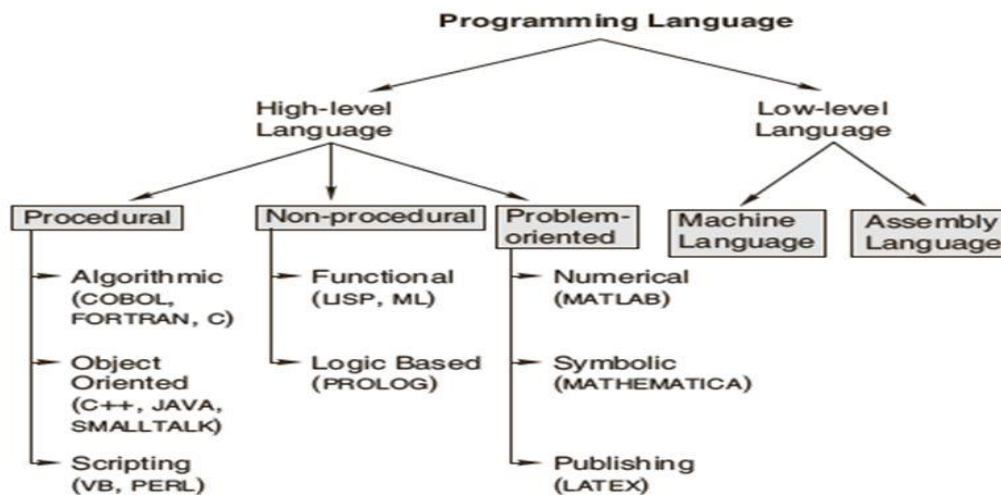
## Programming Languages

Coded language used by programmers to write instructions that a computer can understand to do what the programmer (or the computer user) wants.

classification of programming languages:

- ✓ High level programming language.
- ✓ Low level programming language.

## CLASSIFICATION OF PROGRAMMING LANGUAGES



The first programming language for a computer was Plankalkül, developed by Konrad Zuse for the Z3 between 1943 and 1945. However, it was not implemented until 1998.

Short Code, which was proposed by John Mauchly in 1949, is considered to be the first high-level programming language.

It was designed to represent mathematical expressions in a format readable by human beings.

However, because it had to be translated into machine code before it could be executed, it had relatively slow processing speeds.

## Learn C Programming

C is a powerful general-purpose programming language.

It is fast, portable and available in all platforms.

If you are new to programming, C is a good choice to start your programming journey.

### **What is C (Programming Language)? - The Basics**

- ✓ Before getting started with C programming, let's get familiarized with the language first.
- ✓ C is a general-purpose programming language used for a wide range of applications from Operating systems like Windows and iOS to software that is used for creating 3D movies.
- ✓ C programming is highly efficient. That's the main reason why it's very popular despite being more than 40 years old.
- ✓ Standard C programs are portable. The source code written in one system works in another operating system without any change.
- ✓ As mentioned, it's a good language to start learning programming. If you know C programming, you will not just understand how your program works, but will also be able to create a mental picture on how a computer works.

### **History of C programming**

C is closely associated with Unix Operating system

### **Development of Unix System**

The PDP-11 version of Unix system was written in assembly language. Assembly languages are low-level programming languages that are specific to a particular computer architecture. They are hard to write and understand.

The developers of Unix Operating system (including Dennis Ritchie and Stephen C. Johnson) decided to rewrite the system in B language. However, B couldn't suffice some of the features of PDP-11, which led to the development of C.

In 1972, the development of C started on the PDP-11 Unix system. A large part of Unix was then rewritten in C. By 1973, C was powerful enough to be used in Unix Kernel. Dennis Ritchie and Stephen C. Johnson made further changes to the language for several years to make it portable in Unix Operating system.

### **First Book on C Programming**

In 1978, the first book of C programming, The C Programming Language, was published. The first edition of the book provided programmers informal specification of the language. Written by Brian Kernighan and Dennis Ritchie, this book is popular among C programmers as "K&R".

### **ANSI C**

With the rapid growth of C language for several years, it was time for language to get it standardized.

C89. The first standard of C was published by American National Standards Institute (ANSI) in 1989. This version is commonly popular as C89.

C99. In late 1990's, several new features like inline functions, several new data types and flexible array-members were added to the C standard. This is commonly known as C99.

C11. The C11 standard has new features like type generic macros, atomic operations, anonymous structures that doesn't exist in C99.

All these three standards are also known by the name of ANSI C.

“Standard C programs are portable”. This means, the programs that follow ANSI C standard are portable among operating systems.

If you are new to programming, it's advisable to follow the standard (ANSI C in case of C programming) that is accepted everywhere. It will help you learn the language the way it was intended.

## Features of C Programming Language

### Why Should you learn C programming?

If only it were possible to answer this question with a simple “yes” or “no”. Unfortunately, it's not an easy question to answer and varies from person to person.

Personally speaking, I love C programming. It is a good language to start your programming journey if you are a newbie. Even if you are an experienced programmer, I recommend you to learn it at some point; it will certainly help.

### What will you gain if you learn C?

If you don't know C, you don't know what you are doing as a programmer. Sure, your application works fine and all. But, if you can't say why while `(*s++ = *p++);` copies a string, you're programming on a superstition. ( Joel Spolsky's words, not mine ).

You will understand how a computer works.

If you know C, you will not only know how your program works but, you will be able to create a mental model on how a computer works (including memory management and allocation). You will learn to appreciate the freedom that C provides unlike Python and Java.

Understanding C allows you to write programs that you never thought were possible before (or at the very least, you will have a broader understanding of computer architecture and programming as a whole).

C is the lingua franca of programming.

Almost all high-level programming languages like Java, Python, JavaScript etc. can interface with C programming. Also, it's a good language to express common ideas in programming. Doesn't matter if the person you are talking with doesn't know C, you can still convey your idea in a way they can understand.

Opportunity to work on open source projects that impact millions of people.

At first, you may overlook the fact that C is an important language. If you need to develop a mobile app, you need Java (for Android), Swift and Objective C (for iOS). And there are dozens of languages like C#, PHP, ASP.net, Ruby, Python for building web application. Then, where is C programming?

Python is used for making wide range for applications. And, C is used for making Python. If you want to contribute to Python, you need to know C programming to work on Python interpreter that impacts millions of Python programmers. This is just one example. A large number of softwares that you use today is powered by C.

Some of the larger open source projects where C programming is used are Linux Kernel, Python Interpreter, SQLite Database.

Another language that's commonly used for large open source project is C++. If you know C and C++, you can contribute to large open source projects that impacts hundreds of millions of people.

You will write better programs.

To be honest, this statement may not be true all the time. However, knowing how computer works and manage memory gives you insight on how to write efficient code in other programming languages.

You will find it much easier to learn other programming languages.

A lot of popular programming languages are based on C (and C++, considered superset of C programming with OOP features). If you know C, you will get a head start learning C++.

Languages like C# and Java are related to C and C++. Also, the syntax of JavaScript and PHP is similar to C.

If you know C and C++ programming, you will not have any problem switching to another language.

### **Reasons not to learn C programming**

You can create awesome softwares without knowing C programming at all. Jeff Atwood, one of the creators of Stackoverflow.com, apparently doesn't know C and Stack Overflow is a really good web application.

If you are busy and don't want to invest time on something that doesn't have direct effect on your day-to-day work, C programming is not for you.

Also, if you are a newbie and want to start learning programming with an easier language (C is not the easiest of language to learn), you can start with Python.

Verdict on whether to learn C programming or not

### **For newbie:**

For many, C programming is the best language to start learning programming. However, if you want to start with an easier language which is clean and easier to grasp, go for Python.

### **For experienced programmers:**

It's not absolutely essential but there are perks of learning C programming.

Don't leave your current project immediately (I know you won't) to learn C. You can learn it when you have free time and want to expand your programming skills.

I believe, it's not necessary to learn C immediately. However, you should learn C eventually.



Compile and run C programming on your OS

There are numerous compilers and text editors you can use to run C programming. These compilers and text editors may differ from system to system. You will find the easiest way to run C programming on your computer (Windows, Mac OS X or Linux) in this section.

### **Run C Programming in Windows (XP, 7, 8 and 10)**

To run C Programming in Windows, download a software called Code::Blocks. Then, write C code, save the file with .c extension and execute the code.

To make this procedure even easier, follow this step by step guide.

Go to the binary release download page of Code:Blocks official site.

Under Windows XP / Vista / 7 / 8.x / 10 section, click the link with mingw-setup (highlighted row) either from Sourceforge.net or FossHub.

Download Code::Blocks in Windows

Open the Code::Blocks Setup file and follow the instructions (Next > I agree > Next > Install); you don't need to change anything. This installs the Code::Blocks with gnu gcc compiler, which is the best compiler to start with for beginners.

Now, open Code::Blocks and go to File > New > Empty file (Shortcut: Ctrl + Shift + N)

Create empty file in Codeblocks

Write the C code and save the file with .c extension. To save the file, go to File > Save (Shortcut: Ctrl + S).

Important: The filename should end with a .c extension, like: hello.c, your-program-name.c

Create file with .c extension in Codeblocks to run C programming

To run the program, go to Build > Build and Run (Shortcut: F9). This will build the executable file and run it.

If your program doesn't run and if you see error message "can't find compiler executable in your search path(GNU GCC compiler)", go to Settings > Compiler > Toolchain executables and click Auto-detect. This should solve the issue in most cases.

### **The fun begins: Your first C program**

You will learn to write a "Hello, World!" program in this section.

#### **Why "Hello, World!" program?**

"Hello, World!" is a simple program that displays "Hello, World!" on the screen. Since, it's a very simple program, it is used to illustrate the basic syntax of any programming language.

This program is often used to introduce programming language to a beginner. So, let's get started.

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

```
int main()
{
    printf("Hello, World!\n");
    return 0;
}
```

### How “Hello, World!” program works?

Include stdio.h header file in your program.

C programming is small and cannot do much by itself. You need to use libraries that are necessary to run the program. The stdio.h is a header file and C compiler knows the location of that file. To use the file, you need to include it in your program using #include preprocessor.

### Why do you need stdio.h file in this program?

In this program, we have used printf() function which displays the text inside the quotation mark. Since printf() is defined in stdio.h, you need to include stdio.h.

### The main() function

In C programming, the code execution begins from the start of main() function (doesn't matter if main() isn't located at the beginning).

The code inside the curly braces { } is the body of main() function. The main() function is mandatory in every C program.

```
int main() {
}
```

This program doesn't do anything but, it's a valid C program.

### The printf() function

The printf() is a library function that sends formatted output to the screen (displays the string inside the quotation mark). Notice the semicolon at the end of the statement.

In our program, it displays Hello, World! on the screen.

Remember, you need to include stdio.h file in your program for this to work.

## The return statement

The return statement `return 0;` inside the `main()` function ends the program. This statement isn't mandatory. However, it's considered good programming practice to use it.

Key notes to take away

- ✓ All C program starts from the `main()` function and it's mandatory.
- ✓ You can use the required header file that's necessary in the program.
- ✓ For example: To use `sqrt()` function to calculate square root and `pow()` function to find power of a number, you need to include `math.h` header file in your program.
- ✓ C is case-sensitive; the use of uppercase letter and lowercase letter have different meanings.
- ✓ The C program ends when the program encounters the return statement inside the `main()` function. However, return statement inside the main function is not mandatory.
- ✓ The statement in a C program ends with a semicolon.

## Always follow good programming practice

- ✓ Good programming practice are the informal rules which can improve quality and decrease development time of the software.
- ✓ Some of the programming practices mentioned here are valid in all programming languages whereas some are valid only for C programming.
- ✓ Be consistent with the formatting.
- ✓ The number of spaces you use in the program doesn't matter in C. However, that doesn't mean you should use different number of spaces at different places. Also, use proper indentation so that the code is easier to understand.
- ✓ Use one statement per line.

## What's wrong with the following code?

```
int count;  
float squareRoot = 10.0;  
printf("Square root = %f", squareRoot);
```

Actually, the code is perfectly valid. But, wouldn't this be better:

```
int count;  
float squareRoot = 10.0;  
printf("Square root = %f", squareRoot);
```

The goal here is to write code that your fellow programmers can understand.

Naming convention and Consistency!

Give a proper name to variables and functions and be consistent with it.

```
int a, b;
```

Here, a and b are two variables and I have no idea what they do. Instead you can choose name like:

```
int counter, power;
```

Also, follow a convention while naming. For example:

```
int base_number, powerNumber;
```

**Both conventions:** using \_ to separate words and using capital letter after first word is popular. However, don't use both in one program; choose one and be consistent with it.

### Start Habit of Using Comments

Comments are part of code that compiler ignores.

You can use comments in your program to explain what you are trying to achieve in your program. This helps your fellow programmer to understand the code.

You can use comments in C programming by using //. For example:

```
// My first C program
#include <stdio.h>

int main()
{
    printf("Hello, World!\n"); // displays Hello, World! on the screen
    return 0;
}
```

“Commenting your code is like cleaning your bathroom - you never want to do it, but it really does create a more pleasant experience for you and your guests.”

— Ryan Campbell

### C Programming Keywords and Identifiers

Reserved words in C programming that are part of the syntax. Also, you will learn about identifiers and proper way to name a variable.

### Character set

Character set is a set of alphabets, letters and some special characters that are valid in C language.

### Alphabets

Uppercase: A B C ..... X Y Z

Lowercase: a b c ..... x y z

C accepts both lowercase and uppercase alphabets as variables and functions.

### Digits

0 1 2 3 4 5 6 7 8 9

### Special Characters

Special Characters in C Programming				
,	<	>	.	-
(	)	;	\$	:
%	[	]	#	?
'	&	{	}	"
^	!	*	/	
-	\	~	+	

### White space Characters

blank space, new line, horizontal tab, carriage return and form feed

### C Keywords

Keywords are predefined, reserved words used in programming that have special meanings to the compiler. Keywords are part of the syntax and they cannot be used as an identifier. For example:

```
int money;
```

Here, *int* is a keyword that indicates '*money*' is a variable of type integer.

As C is a case sensitive language, all keywords must be written in lowercase. Here is a list of all keywords allowed in ANSI C.

Keywords in C Language

auto	double	Int	struct
break	else	Long	switch
case	enum	register	typedef
char	extern	Return	union
continue	for	Signed	void
do	if	static	while
default	goto	Sizeof	volatile
const	float	short	unsigned

Along with these keywords, C supports other numerous keywords depending upon the compiler..

## C Identifiers

Identifier refers to name given to entities such as variables, functions, structures etc.

Identifier must be unique. They are created to give unique name to a entity to identify it during the execution of the program. For example:

```
int money;
```

```
double accountBalance;
```

Here, *money* and *accountBalance* are identifiers.

Also remember, identifier names must be different from keywords. You cannot use *int* as an identifier because *int* is a keyword.

### Rules for writing an identifier

1. A valid identifier can have letters (both uppercase and lowercase letters), digits and underscores.
2. The first letter of an identifier should be either a letter or an underscore. However, it is discouraged to start an identifier name with an underscore.
3. There is no rule on length of an identifier. However, the first 31 characters of identifiers are discriminated by the compiler.

## Variables

In programming, a variable is a container (storage area) to hold data.



To indicate the storage area, each variable should be given a unique name (identifier). Variable names are just the symbolic representation of a memory location. For example:

```
int playerScore = 95;
```

Here, *playerScore* is a variable of integer type. The variable is assigned value: 95.

The value of a variable can be changed, hence the name 'variable'.

In C programming, you have to declare a variable before you can use it.

### Rules for naming a variable in C

1. A variable name can have letters (both uppercase and lowercase letters), digits and underscore only.
2. The first letter of a variable should be either a letter or an underscore. However, it is discouraged to start variable name with an underscore. It is because variable name that starts with an underscore can conflict with system name and may cause error.
3. There is no rule on how long a variable can be. However, only the first 31 characters of a variable are checked by the compiler. So, the first 31 letters of two variables in a program should be different.

C is a strongly typed language. What this means is that, the type of a variable cannot be changed.

**Constants/Literals:** A constant is a value or an identifier whose value cannot be altered in a program. For example: 1, 2.5, "C programming is easy", etc.

As mentioned, an identifier also can be defined as a constant.

```
const double PI = 3.14
```

Here, *PI* is a constant. Basically what it means is that, *PI* and *3.14* is same for this program.

Below are the different types of constants you can use in C.

#### 1. Integer constants

An integer constant is a numeric constant (associated with number) without any fractional or exponential part. There are three types of integer constants in C programming:

- decimal constant(base 10)
- octal constant(base 8)
- hexadecimal constant(base 16)

#### For example:

Decimal constants: 0, -9, 22 etc

Octal constants: 021, 077, 033 etc

Hexadecimal constants: 0x7f, 0x2a, 0x521 etc

In C programming, octal constant starts with a 0 and hexadecimal constant starts with a 0x.

## 2. Floating-point constants

A floating point constant is a numeric constant that has either a fractional form or an exponent form. For example:

-2.0

0.0000234

-0.22E-5

## 3. Character constants

A character constant is a constant which uses single quotation around characters. For example: 'a', 'l', 'm', 'F'

## 4. Escape Sequences

Sometimes, it is necessary to use characters which cannot be typed or has special meaning in C programming. For example: newline(enter), tab, question mark etc. In order to use these characters, escape sequence is used.

For example: \n is used for newline. The backslash ( \ ) causes "escape" from the normal way the characters are interpreted by the compiler.

### Escape Sequences

Escape Sequences	Character
\b	Backspace
\f	Form feed
\n	Newline
\r	Return
\t	Horizontal tab
\v	Vertical tab
\\	Backslash
\'	Single quotation mark
\"	Double quotation mark
\?	Question mark

to success...

## Escape Sequences

Escape Sequences	Character
------------------	-----------

\0	Null character
----	----------------

### 5. String constants

String constants are the constants which are enclosed in a pair of double-quote marks. For example:

```
"good"           //string constant
""              //null string constant
"      "        //string constant of six white space
"x"             //string constant having single character.
"Earth is round\n" //prints string with newline.
```

### 6. Enumeration constants

Keyword enum is used to define enumeration types. For example:

```
enum color {yellow, green, black, white};
```

Here, *color* is a variable and *yellow*, *green*, *black* and *white* are the enumeration constants having value 0, 1, 2 and 3 respectively.

## C Programming Data Types

In C programming, variables or memory locations should be declared before it can be used. Similarly, a function also needs to be declared before use.

Data types simply refers to the type and size of data associated with variables and functions.

### Data types in C

#### Fundamental Data Types

- ✓ Integer types
- ✓ Floating type
- ✓ Character type

#### Derived Data Types

- ✓ Arrays
- ✓ Pointers

- ✓ Structures
- ✓ Enumeration

### **Int - Integer data types**

Integers are whole numbers that can have both positive and negative values but no decimal values. Example: 0, -5, 10

In C programming, keyword `int` is used for declaring integer variable. For example:

```
int id;
```

Here, `id` is a variable of type integer.

You can declare multiple variable at once in C programming. For example:

```
int id, age;
```

The size of `int` is either 2 bytes(In older PC's) or 4 bytes. If you consider an integer having size of 4 byte( equal to 32 bits), it can take 232 distinct states as:  $-2^{31}, -2^{31}+1, \dots, -2, -1, 0, 1, 2, \dots, 2^{31}-2, 2^{31}-1$

Similarly, `int` of 2 bytes, it can take 216 distinct states from  $-2^{15}$  to  $2^{15}-1$ . If you try to store larger number than  $2^{31}-1$ , i.e., +2147483647 and smaller number than  $-2^{31}$ , i.e., -2147483648, program will not run correctly.

### **float - Floating types**

Floating type variables can hold real numbers such as: 2.34, -9.382, 5.0 etc. You can declare a floating point variable in C by using either `float` or `double` keyword. For example:

```
float accountBalance;
```

```
double bookPrice;
```

Here, both `accountBalance` and `bookPrice` are floating type variables.

In C, floating values can be represented in exponential form as well. For example:

```
float normalizationFactor = 22.442e2;
```

### **Difference between float and double**

The size of `float` (single precision float data type) is 4 bytes. And the size of `double` (double precision float data type) is 8 bytes. Floating point variables has a precision of 6 digits whereas the precision of `double` is 14 digits.

### **char - Character types**

Keyword `char` is used for declaring character type variables. For example:

```
char test = 'h';
```

Here, test is a character variable. The value of test is 'h'.

The size of character variable is 1 byte.

## C Qualifiers

Qualifiers alters the meaning of base data types to yield a new data type.

### Size qualifiers

Size qualifiers alters the size of a basic type. There are two size qualifiers, long and short. For example:

```
long double i;
```

The size of double is 8 bytes. However, when long keyword is used, that variable becomes 10 bytes.

Learn more about long keyword in C programming.

There is another keyword short which can be used if you previously know the value of a variable will always be a small number.

### Sign qualifiers

Integers and floating point variables can hold both negative and positive values. However, if a variable needs to hold positive value only, unsigned data types are used. For example:

```
// unsigned variables cannot hold negative value
```

```
unsigned int positiveInteger;
```

There is another qualifier signed which can hold both negative and positive only. However, it is not necessary to define variable signed since a variable is signed by default.

An integer variable of 4 bytes can hold data from -231 to 231-1. However, if the variable is defined as unsigned, it can hold data from 0 to 232-1.

It is important to note that, sign qualifiers can be applied to int and char types only.

### Constant qualifiers

An identifier can be declared as a constant. To do so const keyword is used.

```
const int cost = 20;
```

The value of cost cannot be changed in the program.

### Volatile qualifiers

A variable should be declared volatile whenever its value can be changed by some external sources outside the program. Keyword volatile is used for creating volatile variables.

## C Programming Input Output (I/O): printf() and scanf()

C programming has several in-built library functions to perform input and output tasks.

Two commonly used functions for I/O (Input/Output) are printf() and scanf().

The scanf() function reads formatted input from standard input (keyboard) whereas the printf() function sends formatted output to the standard output (screen).

### Example #1: C Output

```
#include <stdio.h> //This is needed to run printf() function.int main(){
    printf("C Programming"); //displays the content inside quotation
    return 0;}
```

**Output:**C Programming

#### How this program works?

- All valid C program must contain the main() function. The code execution begins from the start of main() function.
- The printf() is a library function to send formatted output to the screen. The printf() function is declared in "stdio.h" header file.
- Here, stdio.h is a header file (standard input output header file) and #include is a preprocessor directive to paste the code from the header file when necessary. When the compiler encounters printf() function and doesn't find stdio.h header file, compiler shows error.
- The return 0; statement is the "Exit status" of the program. In simple terms, program ends.

### Example #2: C Integer Output

```
#include <stdio.h>int main(){
    int testInteger = 5;
    printf("Number = %d", testInteger);
    return 0;}
```

#### Output

Number = 5

Inside the quotation of printf() function, there is a format string "%d" (for integer). If the format string matches the argument (*testInteger* in this case), it is displayed on the screen.

### Example #3: C Integer Input/Output

```
#include <stdio.h>int main(){
    int testInteger;
    printf("Enter an integer: ");
    scanf("%d",&testInteger);
    printf("Number = %d",testInteger);
    return 0;}
```

#### Output

Enter an integer: 4



Number = 4

The scanf() function reads formatted input from the keyboard. When user enters an integer, it is stored in variable *testInteger*.

Note the '&' sign before *testInteger*; *&testInteger* gets the address of *testInteger* and the value is stored in that address.

### Example #3: C Floats Input/Output

```
#include <stdio.h>int main(){
    float f;
    printf("Enter a number: ");// %f format string is used in case of floats
    scanf("%f",&f);
    printf("Value = %f", f);
    return 0;}
```

#### Output

```
Enter a number: 23.45
Value = 23.450000
```

The format string "%f" is used to read and display formatted in case of floats.

### Example #4: C Character I/O

```
#include <stdio.h>int main(){
    char chr;
    printf("Enter a character: ");
    scanf("%c",&chr);
    printf("You entered %c.",chr);
    return 0;}
```

#### Output

```
Enter a character: g
You entered g.
```

Format string %c is used in case of character types.

### Little bit on ASCII code

When a character is entered in the above program, the character itself is not stored. Instead, a numeric value(ASCII value) is stored.

And when we displayed that value using "%c" text format, the entered character is displayed.

### Example #5: C ASCII Code

```
#include <stdio.h>int main(){
    char chr;
    printf("Enter a character: ");
```

```
scanf("%c",&chr);

// When %c text format is used, character is displayed in case of character types
printf("You entered %c.\n",chr);

// When %d text format is used, integer is displayed in case of character types
printf("ASCII value of %c is %d.", chr, chr);
return 0;}
```

### Output

Enter a character: g

You entered g.

ASCII value of g is 103.

The ASCII value of character 'g' is 103. When, 'g' is entered, 103 is stored in variable var1 instead of g.

You can display a character if you know ASCII code of that character. This is shown by following example.

### Example #6: C ASCII Code

```
#include <stdio.h>int main(){
    int chr = 69;
    printf("Character having ASCII value 69 is %c.",chr);
    return 0;}
```

### Output

Character having ASCII value 69 is E.

### More on Input/Output of floats and Integers

Integer and floats can be displayed in different formats in C programming.

### Example #7: I/O of Floats and Integers

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

    int integer = 9876;
    float decimal = 987.6543;

    // Prints the number right justified within 6 columns
    printf("4 digit integer right justified to 6 column: %6d\n", integer);

    // Tries to print number right justified to 3 digits but the number is not right adjusted because there are
    // only 4 numbers
    printf("4 digit integer right justified to 3 column: %3d\n", integer);
```

```
// Rounds to two digit places
printf("Floating point number rounded to 2 digits: %.2f\n",decimal);

// Rounds to 0 digit places
printf("Floating point number rounded to 0 digits: %.f\n",987.6543);

// Prints the number in exponential notation(scintific notation)
printf("Floating point number in exponential form: %e\n",987.6543);
return 0;}
```

### Output

4 digit integer right justified to 6 column: 9876  
4 digit integer right justified to 3 column: 9876  
Floating point number rounded to 2 digits: 987.65  
Floating point number rounded to 0 digits: 988  
Floating point number in exponential form: 9.876543e+02

### C Programming Input Output (I/O): printf() and scanf()

There are two in-built functions printf() and scanf() to perform I/O task in C programming. Also, you will learn to write a valid program in C.

C programming has several in-built library functions to perform input and output tasks.

Two commonly used functions for I/O (Input/Output) are printf() and scanf().

The scanf() function reads formatted input from standard input (keyboard) whereas the printf() function sends formatted output to the standard output (screen).

### Example #1: C Output

```
#include <stdio.h> //This is needed to run printf() function.int main(){
    printf("C Programming"); //displays the content inside quotation
    return 0;}
```

### Output

C Programming

### How this program works?

- All valid C program must contain the main() function. The code execution begins from the start of main() function.
- The printf() is a library function to send formatted output to the screen. The printf() function is declared in "stdio.h" header file.
- Here, stdio.h is a header file (standard input output header file) and #include is a preprocessor directive to paste the code from the header file when necessary. When the compiler encounters printf() function and doesn't find stdio.h header file, compiler shows error.
- The return 0; statement is the "Exit status" of the program. In simple terms, program ends.

### Example #2: C Integer Output

```
#include <stdio.h>int main(){
    int testInteger = 5;
    printf("Number = %d", testInteger);
    return 0;}
```

#### Output

Number = 5

Inside the quotation of printf() function, there is a format string "%d" (for integer). If the format string matches the argument (*testInteger* in this case), it is displayed on the screen.

### Example #3: C Integer Input/Output

```
#include <stdio.h>int main(){
    int testInteger;
    printf("Enter an integer: ");
    scanf("%d",&testInteger);
    printf("Number = %d",testInteger);
    return 0;}
```

#### Output

Enter an integer: 4  
Number = 4

The scanf() function reads formatted input from the keyboard. When user enters an integer, it is stored in variable *testInteger*.

Note the '&' sign before *testInteger*; *&testInteger* gets the address of *testInteger* and the value is stored in that address.

### Example #3: C Floats Input/Output

```
#include <stdio.h>int main(){
    float f;
    printf("Enter a number: ");// %f format string is used in case of floats
    scanf("%f",&f);
    printf("Value = %f", f);
    return 0;}
```

#### Output

Enter a number: 23.45  
Value = 23.450000

The format string "%f" is used to read and display formatted in case of floats.

### Example #4: C Character I/O

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

```
printf("Enter a character: ");
scanf("%c",&chr);
printf("You entered %c.",chr);
return 0;}
```

### Output

Enter a character: g  
You entered g.

Format string %c is used in case of character types.

### Little bit on ASCII code

When a character is entered in the above program, the character itself is not stored. Instead, a numeric value(ASCII value) is stored.

And when we displayed that value using "%c" text format, the entered character is displayed.

### Example #5: C ASCII Code

```
#include <stdio.h>int main(){
    char chr;
    printf("Enter a character: ");
    scanf("%c",&chr);

    // When %c text format is used, character is displayed in case of character types
    printf("You entered %c.\n",chr);

    // When %d text format is used, integer is displayed in case of character types
    printf("ASCII value of %c is %d.", chr, chr);
    return 0;}
```

### Output

Enter a character: g  
You entered g.  
ASCII value of g is 103.

The ASCII value of character 'g' is 103. When, 'g' is entered, 103 is stored in variable var1 instead of g.

You can display a character if you know ASCII code of that character. This is shown by following example.

your roots to success...

### Example #6: C ASCII Code

```
#include <stdio.h>int main(){
    int chr = 69;
    printf("Character having ASCII value 69 is %c.",chr);
    return 0;}
```

### Output

Narsimha Reddy Engineering College

NRCM

Character having ASCII value 69 is E.

## More on Input/Output of floats and Integers

Integer and floats can be displayed in different formats in C programming.

### Example #7: I/O of Floats and Integers

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

    int integer = 9876;
    float decimal = 987.6543;

    // Prints the number right justified within 6 columns
    printf("4 digit integer right justified to 6 column: %6d\n", integer);

    // Tries to print number right justified to 3 digits but the number is not right adjusted because there are
    only 4 numbers
    printf("4 digit integer right justified to 3 column: %3d\n", integer);

    // Rounds to two digit places
    printf("Floating point number rounded to 2 digits: %.2f\n",decimal);

    // Rounds to 0 digit places
    printf("Floating point number rounded to 0 digits: %.f\n",987.6543);

    // Prints the number in exponential notation(scientific notation)
    printf("Floating point number in exponential form: %e\n",987.6543);
    return 0;}
```

### Output

```
4 digit integer right justified to 6 column:   9876
4 digit integer right justified to 3 column: 9876
Floating point number rounded to 2 digits: 987.65
Floating point number rounded to 0 digits: 988
Floating point number in exponential form: 9.876543e+02
```

your roots to success...

## C Preprocessor and Macros

In this article, you will be introduced to c preprocessors and you will learn to use #include, #define and conditional compilation.



The C preprocessor is a macro preprocessor (allows you to define macros) that transforms your program before it is compiled. These transformations can be inclusion of header file, macro expansions etc.

All preprocessing directives begins with a # symbol. For example,  
`#define PI 3.14`

Some of the common uses of C preprocessor are:

<u>Include header files</u>
<u>Macros</u>
<u>Conditional Compilation</u>
<u>Diagnostics</u>
<u>Line Control</u>
<u>Pragmas</u>
<u>Other Directives</u>
<u>Preprocessor Output</u>

### Including Header Files

The `#include` preprocessor is used to include header files to a C program. For example,  
`#include <stdio.h>`

Here, "stdio.h" is a header file. The `#include` preprocessor directive replaces the above line with the contents of stdio.h header file which contains function and macro definitions.

That's the reason why you need to use `#include <stdio.h>` before you can use functions like `scanf()` and `printf()`.

You can also create your own header file containing function declaration and include it in your program using this preprocessor directive.

```
#include "my_header.h"
```

### Macros using #define

You can define a macro in C using `#define` preprocessor directive.

A macro is a fragment of code that is given a name. You can use that fragment of code in your program by using the name. For example,

```
#define c 299792458 // speed of light
```

Here, when we use `c` in our program, it's replaced by 3.1415.

### Example 1: Using #define preprocessor

```
#include <stdio.h>#define PI 3.1415
```

```
int main(){  
    float radius, area;  
    printf("Enter the radius: ");  
    scanf("%d", &radius);  
    // Notice, the use of PI  
    area = PI*radius*radius;  
    printf("Area=%.2f",area);  
    return 0;}
```

You can also define macros that works like a function call, known as function-like macros. For example,

```
#define circleArea(r) (3.1415*r*r)
```

Every time the program encounters `circleArea(argument)`, it is replaced by `(3.1415*(argument)*(argument))`.

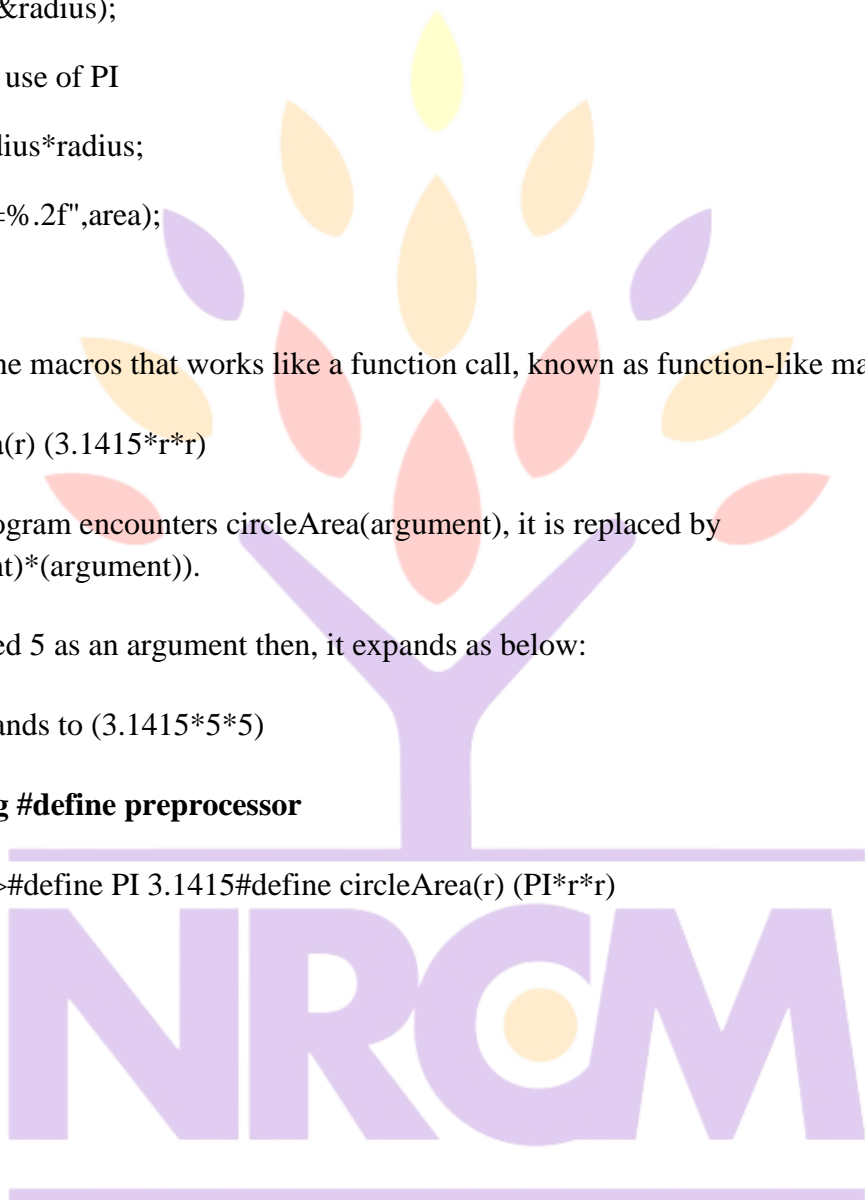
Suppose, we passed 5 as an argument then, it expands as below:

`circleArea(5)` expands to `(3.1415*5*5)`

### Example 2: Using #define preprocessor

```
#include <stdio.h>#define PI 3.1415#define circleArea(r) (PI*r*r)
```

```
int main()  
{  
    int radius;  
    float area;  
  
    printf("Enter the radius: ");  
    scanf("%d", &radius);  
  
    area = circleArea(radius);  
    printf("Area = %.2f", area);  
  
    return 0;
```



your roots to success...

```
}
```

## Conditional Compilation

In C programming, you can instruct preprocessor whether to include certain chunk of code or not. To do so, conditional directives can be used.

It's similar like a if statement. However, there is a big difference you need to understand.

The if statement is tested during the execution time to check whether a block of code should be executed or not whereas, the conditionals is used to include (or skip) certain chunks of code in your program before execution.

## Uses of Conditional

- use different code depending on the machine, operating system
- compile same source file in two different programs
- to exclude certain code from the program but to keep it as reference for future purpose

## How to use conditional?

To use conditional, `#ifdef`, `#if`, `#defined`, `#else` and `#elseif` directives are used.

### `#ifdef` Directive

```
#ifdef MACRO  
    conditional codes  
#endif
```

Here, the conditional codes are included in the program only if `MACRO` is defined.

### `#if`, `#elif` and `#else` Directive

```
#if expression  
    conditional codes  
#endif
```

Here, *expression* is a expression of integer type (can be integers, characters, arithmetic expression, macros and so on). The conditional codes are included in the program only if the *expression* is evaluated to a non-zero value.

The optional `#else` directive can used with `#if` directive.

```
#if expression
    conditional codes if expression is non-zero
#else
    conditional if expression is 0
#endif
```

You can also add nested conditional to your `#if...#else` using `#elif`

```
#if expression
    conditional codes if expression is non-zero
#elif expression1
    conditional codes if expression is non-zero
#elif expression2
    conditional codes if expression is non-zero
... ..
else
    conditional if all expressions are 0
#endif
```

### **#defined**

The special operator `#defined` is used to test whether certain macro is defined or not. It's often used with `#if` directive.

```
#if defined BUFFER_SIZE && BUFFER_SIZE >= 2048
    conditional codes
```

### **Predefined Macros**

There are some predefined macros which are readily for use in C programming.

Predefined macro	Value
<code>__DATE__</code>	String containing the current date
<code>__FILE__</code>	String containing the file name
<code>__LINE__</code>	Integer representing the current line number
<code>__STDC__</code>	If follows ANSI standard C, then value is a nonzero integer
<code>__TIME__</code>	String containing the current date.

### Example #3: predefined Macros

#### C Program to find the current time

```
#include <stdio.h>
int main(){
    printf("Current time: %s",__TIME__); //calculate the current time}
```

#### Output

Current time: 19:54:39

### C Standard Library Functions

In this article, you'll learn about the standard library functions in C. More specifically, what are they, different library functions in C and how to use them in your program.

C Standard library functions or simply C Library functions are inbuilt functions in C programming.

The prototype and data definitions of the functions are present in their respective header files, and must be included in your program to access them.

**For example:** If you want to use `printf()` function, the header file `<stdio.h>` should be included.

```
#include <stdio.h>
int main(){
    // If you use printf() function without including the <stdio.h>
    // header file, this program will show an error.
    printf("Catch me if you can."); }
```

There is at least one function in any C program, i.e., the main() function (which is also a library function). This function is automatically called when your program starts.

### **Advantages of using C library functions**

There are many library functions available in C programming to help you write a good and efficient program. But, why should you use it?

Below are the 4 most important advantages of using standard library functions.

#### **1. They work**

One of the most important reasons you should use library functions is simply because they work.

These functions have gone through multiple rigorous testing and are easy to use.

#### **2. The functions are optimized for performance**

Since, the functions are "standard library" functions, a dedicated group of developers constantly make them better.

In the process, they are able to create the most efficient code optimized for maximum performance.

#### **3. It saves considerable development time**

Since the general functions like printing to a screen, calculating the square root, and many more are already written. You shouldn't worry about creating them once again.

It saves valuable time and your code may not always be the most efficient.

#### **3. The functions are portable**

With ever changing real world needs, your application is expected to work every time, everywhere.

And, these library functions help you in that they do the same thing on every computer.

This saves time, effort and makes your program portable.

### **Use Of Library Function To Find Square root**

However, in C programming you can find the square root by just using sqrt() function which is defined under header file "math.h"

```
#include <stdio.h>#include <math.h>int main(){  
    float num, root;  
    printf("Enter a number: ");
```



```
scanf("%f", &num);  
  
// Computes the square root of num and stores in root.  
  
root = sqrt(num);  
  
printf("Square root of %.2f = %.2f", num, root);  
  
return 0;}
```

When you run the program, the output will be:

Enter a number: 12

Square root of 12.00 = 3.46

### C Library Functions Under Different Header File

#### C Header Files

- <assert.h> Program assertion functions
- <ctype.h> Character type functions
- <locale.h> Localization functions
- <math.h> Mathematics functions
- <setjmp.h> Jump functions
- <signal.h> Signal handling functions
- <stdarg.h> Variable arguments handling functions
- <stdio.h> Standard Input/Output functions
- <stdlib.h> Standard Utility functions
- <string.h> String handling functions
- <time.h> Date time functions

your roots to success...

## Operators:

### C Arithmetic Operators

An arithmetic operator performs mathematical operations such as addition, subtraction and multiplication on numerical values (constants and variables).

Operator	Meaning of Operator
+	addition or unary plus
-	subtraction or unary minus
*	multiplication
/	division
%	remainder after division( modulo division)

### Example #1: Arithmetic Operators

```
// C Program to demonstrate the working of arithmetic operators#include <stdio.h>int main(){
```

```
int a = 9,b = 4, c;
```

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

```
printf("a+b = %d \n",c);
```

```
c = a-b;
```

```
printf("a-b = %d \n",c);
```

```
c = a*b;
```

```
printf("a*b = %d \n",c);
```

```
c=a/b;

printf("a/b = %d \n",c);

c=a%b;

printf("Remainder when a divided by b = %d \n",c);

return 0;}
```

### Output

a+b = 13

a-b = 5

a\*b = 36

a/b = 2

Remainder when a divided by b=1

The operators +, - and \* computes addition, subtraction and multiplication respectively as you might have expected.

In normal calculation,  $9/4 = 2.25$ . However, the output is 2 in the program.

It is because both variables a and b are integers. Hence, the output is also an integer. The compiler neglects the term after decimal point and shows answer 2 instead of 2.25.

The modulo operator % computes the remainder. When a = 9 is divided by b = 4, the remainder is 1. The % operator can only be used with integers.

Suppose a = 5.0, b = 2.0, c = 5 and d = 2. Then in C programming,

a/b = 2.5 // Because both operands are floating-point variables

a/d = 2.5 // Because one operand is floating-point variable

c/b = 2.5 // Because one operand is floating-point variable

c/d = 2 // Because both operands are integers

### Increment and decrement operators

C programming has two operators increment ++ and decrement -- to change the value of an operand (constant or variable) by 1.

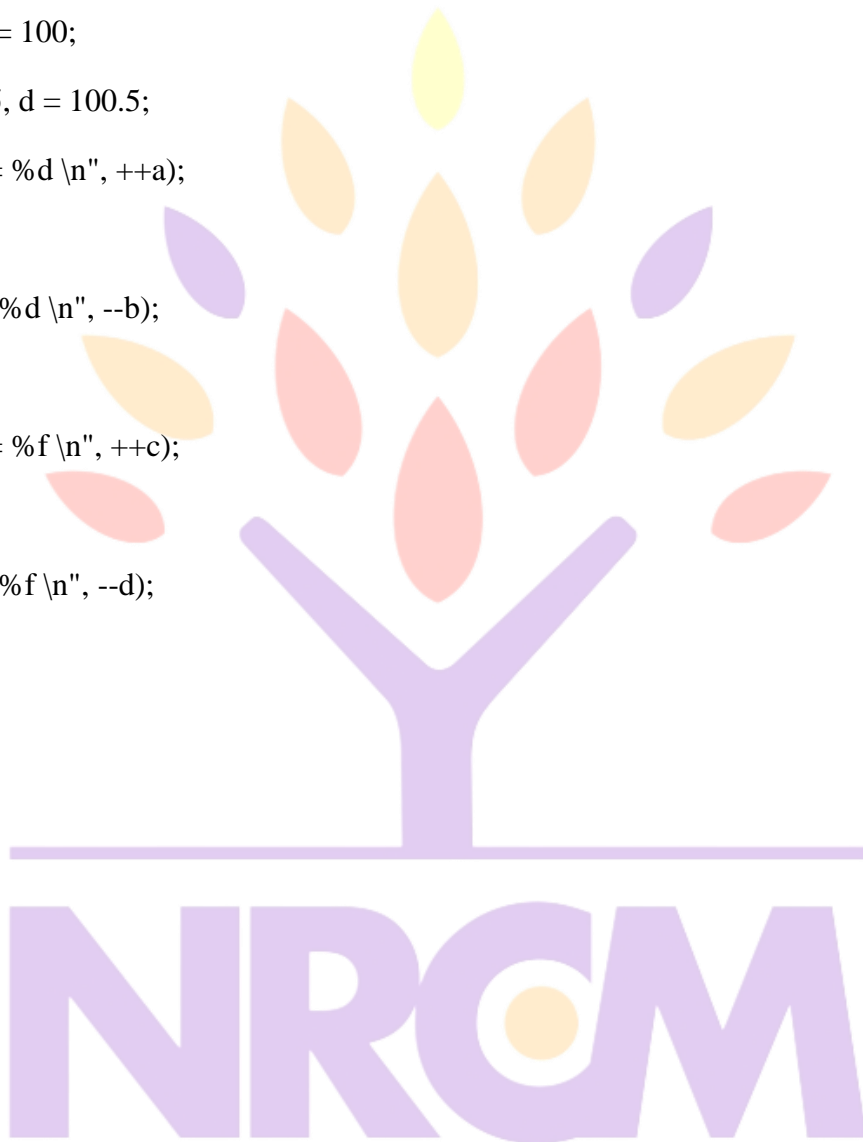
Increment ++ increases the value by 1 whereas decrement -- decreases the value by 1. These two operators are unary operators, meaning they only operate on a single operand.

### Example #2: Increment and Decrement Operators

```
// C Program to demonstrate the working of increment and decrement operators#include <stdio.h>int main(){    int a = 10, b = 100;    float c = 10.5, d = 100.5;    printf("++a = %d \n", ++a);    printf("--b = %d \n", --b);    printf("++c = %f \n", ++c);    printf("--d = %f \n", --d);    return 0;}
```

### Output

```
++a = 11  
--b = 99  
++c = 11.500000  
--d = 99.500000
```



Here, the operators ++ and -- are used as prefix. These two operators can also be used as postfix like a++ and a--. Visit [this page](#) to learn more on how increment and decrement operators work when used as postfix.

your roots to success...

### C Assignment Operators

An assignment operator is used for assigning a value to a variable. The most common assignment operator is =

Operator	Example	Same as
=	a = b	a = b
+=	a += b	a = a+b
-=	a -= b	a = a-b
*=	a *= b	a = a*b
/=	a /= b	a = a/b
%=	a %= b	a = a%b

### Example #3: Assignment Operators

```
// C Program to demonstrate the working of assignment operators#include <stdio.h>int main(){
```

```
int a = 5, c;
```

```
c = a;
```

```
printf("c = %d \n", c);
```

```
c += a; // c = c+a
```

```
printf("c = %d \n", c);
```

```
c -= a; // c = c-a
```

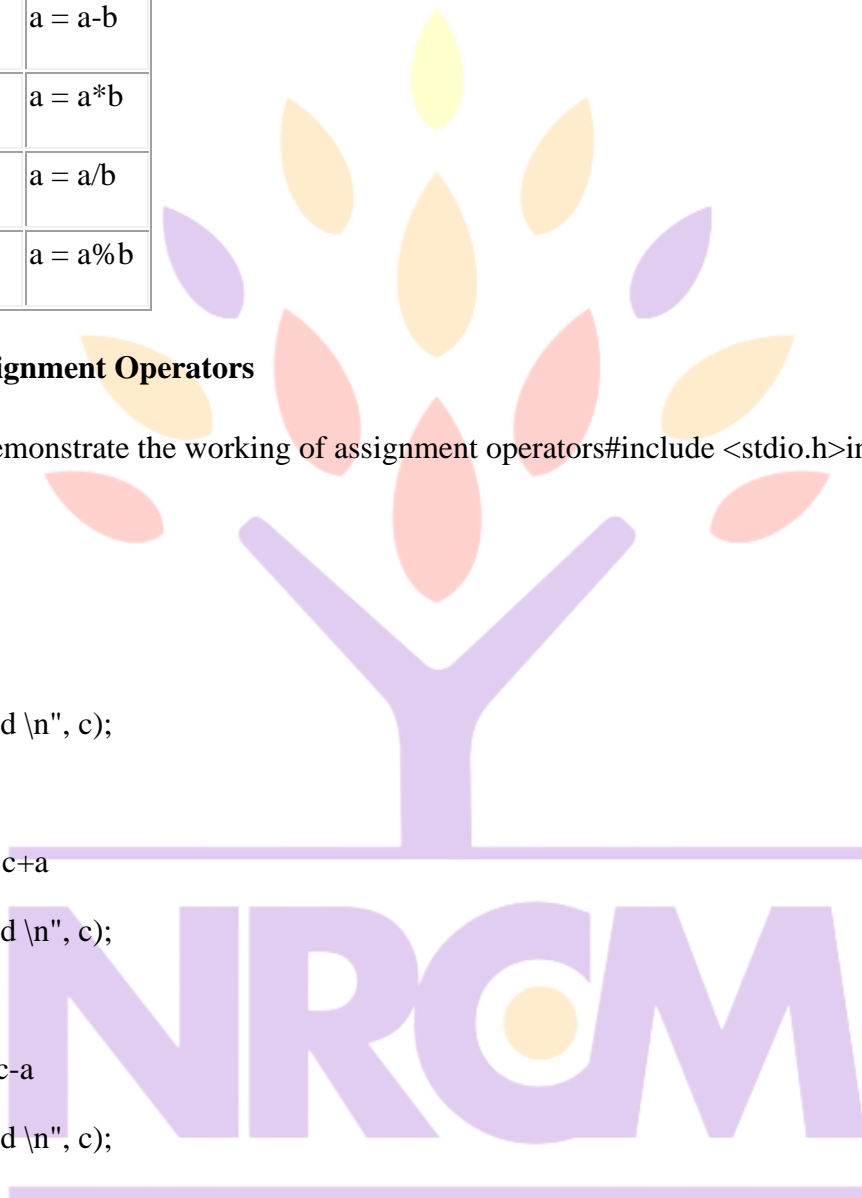
```
printf("c = %d \n", c);
```

```
c *= a; // c = c*a
```

```
printf("c = %d \n", c);
```

```
c /= a; // c = c/a
```

```
printf("c = %d \n", c);
```



your roots to success...

```

c %= a; // c = c%a

printf("c = %d \n", c);

return 0;}

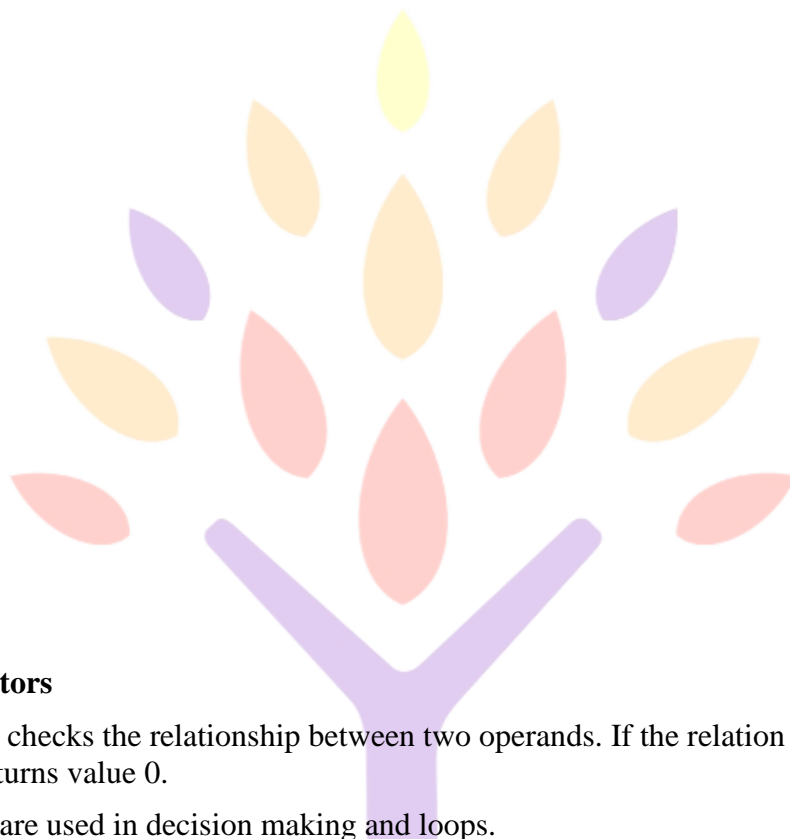
```

## Output

```

c = 5
c = 10
c = 5
c = 25
c = 5
c = 0

```



## C Relational Operators

A relational operator checks the relationship between two operands. If the relation is true, it returns 1; if the relation is false, it returns value 0.

Relational operators are used in decision making and loops.

Operator	Meaning of Operator	Example
==	Equal to	5 == 3 returns 0
>	Greater than	5 > 3 returns 1
<	Less than	5 < 3 returns 0
!=	Not equal to	5 != 3 returns 1
>=	Greater than or equal to	5 >= 3 returns 1
<=	Less than or equal to	5 <= 3 return 0

### Example #4: Relational Operators

```

// C Program to demonstrate the working of arithmetic operators#include <stdio.h>int main(){
    int a = 5, b = 5, c = 10;
    printf("%d == %d = %d \n", a, b, a == b); // true
    printf("%d == %d = %d \n", a, c, a == c); // false
    printf("%d > %d = %d \n", a, b, a > b); //false
    printf("%d > %d = %d \n", a, c, a > c); //false
}

```



```

printf("%d < %d = %d \n", a, b, a < b); //false
printf("%d < %d = %d \n", a, c, a < c); //true
printf("%d != %d = %d \n", a, b, a != b); //false
printf("%d != %d = %d \n", a, c, a != c); //true
printf("%d >= %d = %d \n", a, b, a >= b); //true
printf("%d >= %d = %d \n", a, c, a >= c); //false
printf("%d <= %d = %d \n", a, b, a <= b); //true
printf("%d <= %d = %d \n", a, c, a <= c); //true

```

```
return 0;}
```

### Output

```

5 == 5 = 1
5 == 10 = 0
5 > 5 = 0
5 > 10 = 0
5 < 5 = 0
5 < 10 = 1
5 != 5 = 0
5 != 10 = 1
5 >= 5 = 1
5 >= 10 = 0
5 <= 5 = 1
5 <= 10 = 1

```



### C Logical Operators

An expression containing logical operator returns either 0 or 1 depending upon whether expression results true or false. Logical operators are commonly used in decision making in C programming.

Operator	Meaning of Operator	Example
&&	Logical AND. True only if all operands are true.	If c = 5 and d = 2 then, expression ((c == 5) && (d > 5)) equals to 0.
	Logical OR. True only if either one operand is true	If c = 5 and d = 2 then, expression ((c == 5)    (d > 5)) equals to 1.

Operator	Meaning of Operator	Example
!	Logical NOT. True only if the operand is 0	If c = 5 then, expression !(c == 5) equals to 0.

### Example #5: Logical Operators

// C Program to demonstrate the working of logical operators

```
#include <stdio.h>int main(){
    int a = 5, b = 5, c = 10, result;

    result = (a = b) && (c > b);
    printf("(a = b) && (c > b) equals to %d \n", result);

    result = (a = b) && (c < b);
    printf("(a = b) && (c < b) equals to %d \n", result);

    result = (a = b) || (c < b);
    printf("(a = b) || (c < b) equals to %d \n", result);

    result = (a != b) || (c < b);
    printf("(a != b) || (c < b) equals to %d \n", result);

    result = !(a != b);
    printf("!(a == b) equals to %d \n", result);

    result = !(a == b);
    printf("!(a == b) equals to %d \n", result);

    return 0;}
```

### Output

(a = b) && (c > b) equals to 1

(a = b) && (c < b) equals to 0

(a = b) || (c < b) equals to 1

$(a \neq b) \parallel (c < b)$  equals to 0

$!(a \neq b)$  equals to 1

$!(a == b)$  equals to 0

### Explanation of logical operator program

- $(a = b) \&\& (c > 5)$  evaluates to 1 because both operands  $(a = b)$  and  $(c > 5)$  is 1 (true).
- $(a = b) \&\& (c < b)$  evaluates to 0 because operand  $(c < b)$  is 0 (false).
- $(a = b) \parallel (c < b)$  evaluates to 1 because  $(a = b)$  is 1 (true).
- $(a \neq b) \parallel (c < b)$  evaluates to 0 because both operand  $(a \neq b)$  and  $(c < b)$  are 0 (false).
- $!(a \neq b)$  evaluates to 1 because operand  $(a \neq b)$  is 0 (false). Hence,  $!(a \neq b)$  is 1 (true).
- $!(a == b)$  evaluates to 0 because  $(a == b)$  is 1 (true). Hence,  $!(a == b)$  is 0 (false).

### Bitwise Operators

During computation, mathematical operations like: addition, subtraction, addition and division are converted to bit-level which makes processing faster and saves power.

Bitwise operators are used in C programming to perform bit-level operations.

Operators	Meaning of operators
&	Bitwise AND
	Bitwise OR
^	Bitwise exclusive OR
~	Bitwise complement
<<	Shift left
>>	Shift right

### Other Operators

#### Comma Operator

Comma operators are used to link related expressions together. For example:

```
int a, c = 5, d;
```

#### The sizeof operator

The sizeof is an unary operator which returns the size of data (constant, variables, array, structure etc).

## Example #6: sizeof Operator

```
#include <stdio.h>int main(){
    int a, e[10];
    float b;
    double c;
    char d;
    printf("Size of int=%lu bytes\n",sizeof(a));
    printf("Size of float=%lu bytes\n",sizeof(b));
    printf("Size of double=%lu bytes\n",sizeof(c));
    printf("Size of char=%lu byte\n",sizeof(d));
    printf("Size of integer type array having 10 elements = %lu bytes\n", sizeof(e));
    return 0;}
```

## Output

Size of int = 4 bytes

Size of float = 4 bytes

Size of double = 8 bytes

Size of char = 1 byte

Size of integer type array having 10 elements = 40 bytes

## C-Ternary Operator (?:)

A conditional operator is a ternary operator, that is, it works on 3 operands.

### Conditional Operator Syntax

conditionalExpression ? expression1 : expression2

The conditional operator works as follows:

- The first expression conditionalExpression is evaluated at first. This expression evaluates to 1 if it's and evaluates to 0 if it's false.

- If *conditionalExpression* is true, *expression1* is evaluated.
- If *conditionalExpression* is false, *expression2* is evaluated.

### Example #7: C conditional Operator

```
#include <stdio.h>
int main(){
    char February;
    int days;
    printf("If this year is leap year, enter 1. If not enter any integer: ");
    scanf("%c",&February);
    // If test condition (February == '1') is true, days equal to 29.
    // If test condition (February =='1') is false, days equal to 28.
    days = (February == '1') ? 29 : 28;
    printf("Number of days in February = %d",days);
    return 0;}
```

### Output

If this year is leap year, enter 1. If not enter any integer: 1

Number of days in February = 29

Other operators such as & (reference operator), \* (dereference operator) and -> (member selection) operator will be discussed in C pointers.

### C if, if...else and Nested if...else Statement

Decision making is used to specify the order in which statements are executed. In this tutorial, you will learn to create decisions using different forms of if...else statement.

#### C if statement

```
if (testExpression)
```

```
{
    // statements
}
```

The if statement evaluates the test expression inside parenthesis.

If test expression is evaluated to true (nonzero), statements inside the body of if is executed.

If test expression is evaluated to false (0), statements inside the body of if is skipped.

To learn more on when test expression is evaluated to nonzero (true) and 0 (false), check out relational and logical operators.

### Flowchart of if statement

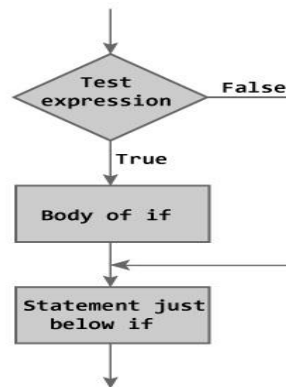


Figure: Flowchart of if Statement

### Example #1: C if statement

// Program to display a number if user enters negative number// If user enters positive number, that number won't be displayed

```
#include <stdio.h>int main(){  
    int number;  
  
    printf("Enter an integer: ");  
    scanf("%d", &number);  
  
    // Test expression is true if number is less than 0  
    if (number < 0)  
    {  
        printf("You entered %d.\n", number);  
    }  
  
    printf("The if statement is easy.");  
  
    return 0;}
```

## Output 1

Enter an integer: -2

You entered -2.

The if statement is easy.

When user enters -2, the test expression ( $\text{number} < 0$ ) becomes true. Hence, You entered -2 is displayed on the screen.

## Output 2

Enter an integer: 5

The if statement in C programming is easy.

When user enters 5, the test expression ( $\text{number} < 0$ ) becomes false and the statement inside the body of if is skipped.

## C if...else statement

The if...else statement executes some code if the test expression is true (nonzero) and some other code if the test expression is false (0).

## Syntax of if...else

```
if (testExpression) {  
    // codes inside the body of if  
}  
else {  
    // codes inside the body of else  
}
```

If test expression is true, codes inside the body of if statement is executed and, codes inside the body of else statement is skipped.

If test expression is false, codes inside the body of else statement is executed and, codes inside the body of if statement is skipped.

## Flowchart of if...else statement



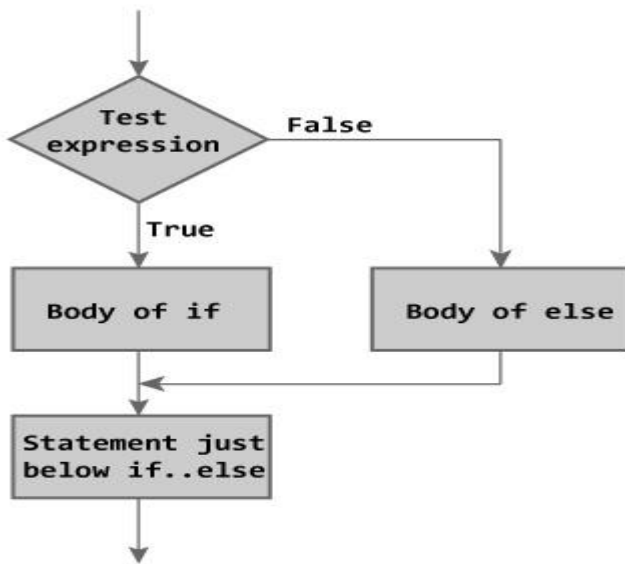


Figure: Flowchart of if...else Statement

### Example #2: C if...else statement

// Program to check whether an integer entered by the user is odd or even

```

#include <stdio.h>int main(){
    int number;
    printf("Enter an integer: ");
    scanf("%d",&number);
    // True if remainder is 0
    if( number%2 == 0 )
        printf("%d is an even integer.",number);
    else
        printf("%d is an odd integer.",number);
    return 0;
}
  
```

### Output

Enter an integer: 7

7 is an odd integer.

When user enters 7, the test expression (  $\text{number} \% 2 == 0$  ) is evaluated to false. Hence, the statement inside the body of else statement `printf("%d is an odd integer");` is executed and the statement inside the body of if is skipped.

### **Nested if...else statement (if...elseif...else Statement)**

The if...else statement executes two different codes depending upon whether the test expression is true or false. Sometimes, a choice has to be made from more than 2 possibilities.

The nested if...else statement allows you to check for multiple test expressions and execute different codes for more than two conditions.

Syntax of nested if...else statement.

```
if (testExpression1)
{
    // statements to be executed if testExpression1 is true
}
else if(testExpression2)
{
    // statements to be executed if testExpression1 is false and testExpression2 is true
}
else if (testExpression 3)
{
    // statements to be executed if testExpression1 and testExpression2 is false and testExpression3 is true
}
.
.
else
{
    // statements to be executed if all test expressions are false
}
```

### **Example #3: C Nested if...else statement**

// Program to relate two integers using =, > or <

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

Narsimha Reddy Engineering College

NRCM

```
int number1, number2;

printf("Enter two integers: ");

scanf("%d %d", &number1, &number2);

//checks if two integers are equal.

if(number1 == number2)
{
    printf("Result: %d = %d",number1,number2);
}

//checks if number1 is greater than number2.

else if (number1 > number2)
{
    printf("Result: %d > %d", number1, number2);
}

// if both test expression is false

else
{
    printf("Result: %d < %d",number1, number2);
}

return 0;}
```

**Output**Enter two integers: 1223

Result: 12 < 23

## C Programming for Loop

Loops are used in programming to repeat a specific block of code. After reading this tutorial, you will learn to create a for loop in C programming.

your roots to success...

Loops are used in programming to repeat a specific block until some end condition is met. There are three loops in C programming:

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

## for Loop

The syntax of for loop is:

```
for (initializationStatement; testExpression; updateStatement)
{
    // codes
}
```

### How for loop works?

The initialization statement is executed only once.

Then, the test expression is evaluated. If the test expression is false (0), for loop is terminated. But if the test expression is true (nonzero), codes inside the body of for loop is executed and the update expression is updated.

This process repeats until the test expression is false.

The *for* loop is commonly used when the number of iterations is known.

### for loop Flowchart

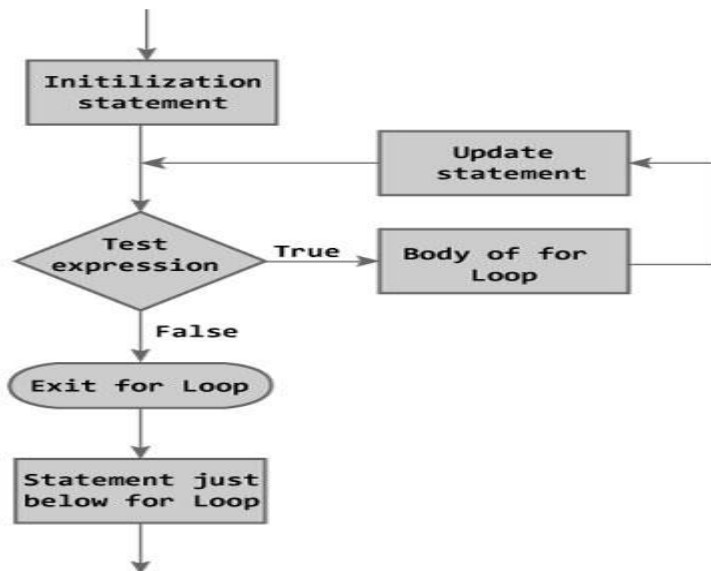


Figure: Flowchart of for Loop

success...

### Example: for loop

```
// Program to calculate the sum of first n natural numbers// Positive integers 1,2,3...n are known as natural numbers
```

```
#include <stdio.h>int main(){
    int num, count, sum = 0;

    printf("Enter a positive integer: ");
    scanf("%d", &num);

    // for loop terminates when n is less than count
    for(count = 1; count <= num; ++count)
    {
        sum += count;
    }

    printf("Sum = %d", sum);

    return 0;}
```

## Output

Enter a positive integer: 10

Sum = 55

The value entered by the user is stored in variable *num*. Suppose, the user entered 10.

The *count* is initialized to 1 and the test expression is evaluated. Since, the test expression *count* <= *num* (1 less than or equal to 10) is true, the body of for loop is executed and the value of *sum* will equal to 1.

Then, the update statement *++count* is executed and *count* will equal to 2. Again, the test expression is evaluated. Since, 2 is also less than 10, the test expression is evaluated to true and the body of for loop is executed. Now, the *sum* will equal 3.

This process goes on and the sum is calculated until the *count* reaches 11.

When the *count* is 11, the test expression is evaluated to 0 (false) as 11 is not less than or equal to 10. Therefore, the loop terminates and next, the total sum is printed.

## C programming while and do...while Loop

Loops are used in programming to repeat a specific block of code. After reading this tutorial, you will learn how to create a while and do...while loop in C programming.

Loops are used in programming to repeat a specific block until some end condition is met. There are three loops in C programming:

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

### **while loop**

The syntax of a while loop is:

```
while (testExpression)
{
    //codes
}
```

where, testExpression checks the condition is true or false before each loop.

### **How while loop works?**

The while loop evaluates the test expression.

If the test expression is true (nonzero), codes inside the body of while loop is evaluated. The test expression is evaluated again. The process goes on until the test expression is false.

When the test expression is false, the while loop is terminated.

### **Flowchart of while loop**

#### **Example #1: while loop**

```
// Program to find factorial of a number
// For a positive integer n, factorial = 1*2*3...n
```

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

```
int main()
```

```
{
```

```

int number;

long long factorial;

printf("Enter an integer: ");

scanf("%d",&number);

factorial = 1;

// loop terminates when number is less than or equal to 0
while (number > 0)
{
    factorial *= number; // factorial = factorial*number;
    --number;
}

printf("Factorial= %lld", factorial);

return 0;
}

```

### Output

```

Enter an integer: 5
Factorial = 120.

```

### do...while loop

The do..while loop is similar to the while loop with one important difference. The body of do...while loop is executed once, before checking the test expression. Hence, the do...while loop is executed at least once.

### do...while loop Syntax

```

do
{
    // codes
}

```



```
while (testExpression);
```

### How do...while loop works?

The code block (loop body) inside the braces is executed once.

Then, the test expression is evaluated. If the test expression is true, the loop body is executed again. This process goes on until the test expression is evaluated to 0 (false).

When the test expression is false (nonzero), the do...while loop is terminated.

### Example #2: do...while loop

```
// Program to add numbers until user enters zero
```

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

```
int main()
```

```
{
```

```
    double number, sum = 0;
```

```
    // loop body is executed at least once
```

```
    do
```

```
    {
```

```
        printf("Enter a number: ");
```

```
        scanf("%lf", &number);
```

```
        sum += number;
```

```
    }
```

```
    while(number != 0.0);
```

```
    printf("Sum = %.2lf",sum);
```

```
    return 0;
```

```
}
```

### Output

Narsimha Reddy Engineering College

NRCM

Enter a number: 1.5

Enter a number: 2.4

Enter a number: -3.4

Enter a number: 4.2

Enter a number: 0

Sum = 4.70

## C Programming break and continue Statement

It is sometimes desirable to skip some statements inside the loop or terminate the loop immediately without checking the test expression.

In such cases, break and continue statements are used.

### break Statement

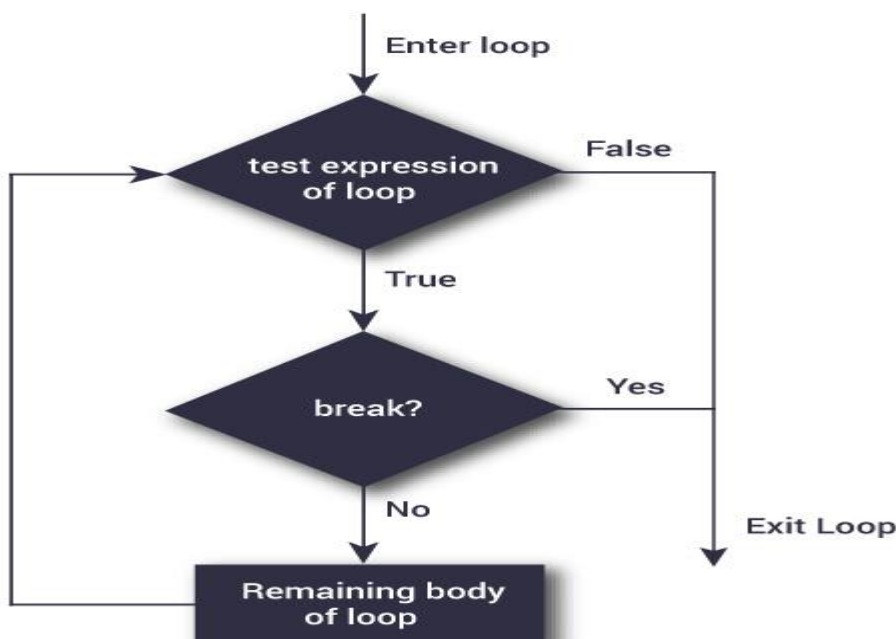
The break statement terminates the loop (for, while and do...while loop) immediately when it is encountered. The break statement is used with decision making statement such as if...else.

### Syntax of break statement

```
break;
```


The simple code above is the syntax for break statement.

### Flowchart of break statement




## How break statement works?

```
while (test Expression)
{
    // codes
    if (condition for break)
    {
        break;
    }
    // codes
}
```



```
for (init, condition, update)
{
    // codes
    if (condition for break)
    {
        break;
    }
    // codes
}
```



### Example #1: break statement

// Program to calculate the sum of maximum of 10 numbers// Calculates sum until user enters positive number

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

```
int i;
```

```
double number, sum = 0.0;
```

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

```
{
```

```
printf("Enter a n%d: ",i);
```

```
scanf("%lf",&number);
```

```
// If user enters negative number, loop is terminated
```

```
if(number < 0.0)
```

```
{
```

your roots to success...

```
break;
```

```
}
```

```
sum += number; // sum = sum + number;
```

```
}
```

```
printf("Sum = %.2lf",sum);
```

```
return 0;}
```

### Output

Enter a n1: 2.4

Enter a n2: 4.5

Enter a n3: 3.4

Enter a n4: -3

Sum = 10.30

This program calculates the sum of maximum of 10 numbers. It's because, when the user enters negative number, the break statement is executed and loop is terminated.

In C programming, break statement is also used with switch...case statement.

### continue Statement

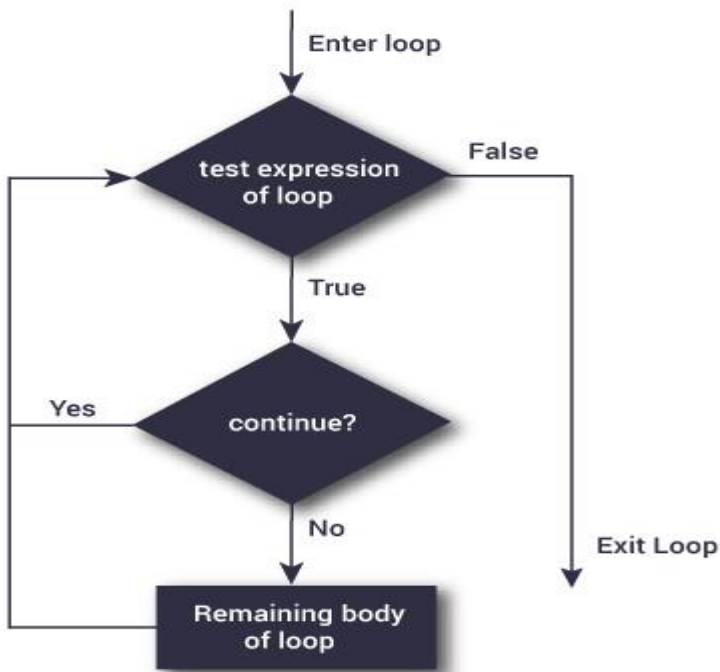
The continue statement skips some statements inside the loop. The continue statement is used with decision making statement such as if...else.

### Syntax of continue Statement

```
continue;
```

### Flowchart of continue Statement

your roots to success...



How continue statement works?

```

while (test Expression)
{
    // codes
    if (condition for continue)
    {
        continue;
    }
    // codes
}
  
```

```

for (init, condition, update)
{
    // codes
    if (condition for continue)
    {
        continue;
    }
    // codes
}
  
```

Example #2: continue statement

// Program to calculate sum of maximum of 10 numbers// Negative numbers are skipped from calculation

```
# include <stdio.h>int main(){  
    int i;  
    double number, sum = 0.0;  
  
    for(i=1; i <= 10; ++i)  
    {  
        printf("Enter a n%d: ",i);  
        scanf("%lf",&number);  
  
        // If user enters negative number, loop is terminated  
        if(number < 0.0)  
        {  
            continue;  
        }  
  
        sum += number; // sum = sum + number;  
    }  
  
    printf("Sum = %.2lf",sum);  
  
    return 0;}
```

### Output

Enter a n1: 1.1

Enter a n2: 2.2

Enter a n3: 5.5

Enter a n4: 4.4

Enter a n5: -3.4

Enter a n6: -45.5

Enter a n7: 34.5

Enter a n8: -4.2

Enter a n9: -1000

Enter a n10: 12

Sum = 59.70

In the program, when the user enters positive number, the sum is calculated using `sum += number;` statement.

When the user enters negative number, the `continue` statement is executed and skips the negative number from calculation.

### **C switch...case Statement**

The `if..else..if` ladder allows you to execute a block code among many alternatives. If you are checking on the value of a single variable in `if...else...if`, it is better to use `switch` statement.

The `switch` statement is often faster than nested `if...else` (not always). Also, the syntax of `switch` statement is cleaner and easy to understand.

### **Syntax of switch...case**

```
switch (n)
{
    case constant1:
        // code to be executed if n is equal to constant1;
        break;
    case constant2:
        // code to be executed if n is equal to constant2;
        break;
    .
    .
    .
    default:
        // code to be executed if n doesn't match any constant
```



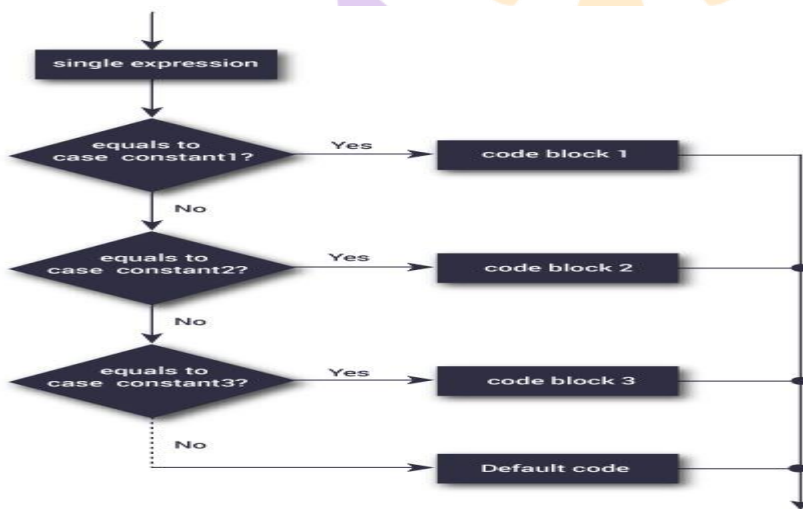
}

When a case constant is found that matches the switch expression, control of the program passes to the block of code associated with that case.

In the above pseudocode, suppose the value of  $n$  is equal to *constant2*. The compiler will execute the block of code associate with the case statement until the end of switch block, or until the break statement is encountered.

The break statement is used to prevent the code running into the next case.

### switch Statement Flowchart



**Example: switch Statement// Program to create a simple calculator// Performs addition, subtraction, multiplication or division depending the input from user**

```
# include <stdio.h>
int main() {
    char operator;
    double firstNumber,secondNumber;
    printf("Enter an operator (+, -, *, /): ");
    scanf("%c", &operator);
    printf("Enter two operands: ");
    scanf("%lf %lf",&firstNumber, &secondNumber);
    switch(operator)
    {
```

```

case '+':
    printf("%.1lf + %.1lf = %.1lf",firstNumber, secondNumber, firstNumber+secondNumber);
    break;

case '-':
    printf("%.1lf - %.1lf = %.1lf",firstNumber, secondNumber, firstNumber-secondNumber);
    break;

case '*':
    printf("%.1lf * %.1lf = %.1lf",firstNumber, secondNumber, firstNumber*secondNumber);
    break;

case '/':
    printf("%.1lf / %.1lf = %.1lf",firstNumber, secondNumber, firstNumber/firstNumber);
    break;

// operator is doesn't match any case constant (+, -, *, /)
default:
    printf("Error! operator is not correct");
}

return 0;}

```

### Output

Enter an operator (+, -, \*, /): -

Enter two operands: 32.5

12.4

32.5 - 12.4 = 20.1

The - operator entered by the user is stored in *operator* variable. And, two operands 32.5 and 12.4 are stored in variables *firstNumber* and *secondNumber* respectively.

Then, control of the program jumps to

```
printf("%.1lf / %.1lf = %.1lf",firstNumber, secondNumber, firstNumber/firstNumber);
```

Finally, the break statement ends the switch statement.

If break statement is not used, all cases after the correct case is executed.

### C goto Statement

The goto statement is used to alter the normal sequence of a C program.

#### Syntax of goto statement

```
goto label;
```

.....

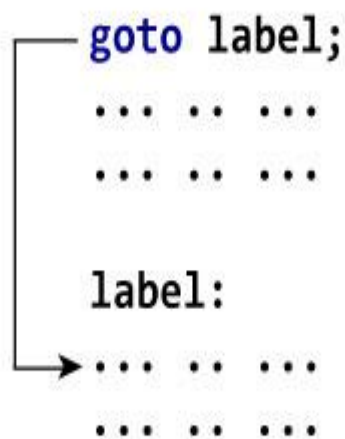
.....

.....

```
label:
```

```
statement;
```

The label is an identifier. When goto statement is encountered, control of the program jumps to label: and starts executing the code.



to success...

## Example: goto Statement

// Program to calculate the sum and average of maximum of 5 numbers// If user enters negative number, the sum and average of previously entered positive number is displayed

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

```
int main(){
```

```
    const int maxInput = 5;
```

```
    int i;
```

```
    double number, average, sum=0.0;
```

```
    for(i=1; i<=maxInput; ++i)
```

```
    {
```

```
        printf("%d. Enter a number: ", i);
```

```
        scanf("%lf",&number);
```

```
    // If user enters negative number, flow of program moves to label jump
```

```
        if(number < 0.0)
```

```
            goto jump;
```

```
        sum += number; // sum = sum+number;
```

```
    }
```

```
    jump:
```

```
    average=sum/(i-1);
```

```
    printf("Sum = %.2f\n", sum);
```

```
    printf("Average = %.2f", average);
```

```
    return 0;}
```

## Output

1. Enter a number: 3
2. Enter a number: 4.3

3. Enter a number: 9.3

4. Enter a number: -2.9

Sum = 16.60

### Reasons to avoid goto statement

The use of goto statement may lead to code that is buggy and hard to follow. For example:

one:

```
for (i = 0; i < number; ++i)
```

```
{
```

```
    test += i;
```

```
    goto two;
```

```
}
```

two:

```
if (test > 5) {
```

```
    goto three;
```

```
}
```

... ..Also, goto statement allows you to do bad stuff such as jump out of scope.

That being said, goto statement can be useful sometimes. For example: to break from nested loops.

### C Programming Functions

A function is a block of code that performs a specific task.

Suppose, a program related to graphics needs to create a circle and color it depending upon the radius and color from the user. You can create two functions to solve this problem:

- create a circle function
- color function

Dividing complex problem into small components makes program easy to understand and use.

### Types of functions in C programming

Depending on whether a function is defined by the user or already included in C compilers, there are two types of functions in C programming

There are two types of functions in C programming:

- Standard library functions
- User defined functions

### Standard library functions

The standard library functions are built-in functions in C programming to handle tasks such as mathematical computations, I/O processing, string handling etc.

These functions are defined in the header file. When you include the header file, these functions are available for use. For example:

The printf() is a standard library function to send formatted output to the screen (display output on the screen). This function is defined in "stdio.h" header file.

There are other numerous library functions defined under "stdio.h", such as scanf(), fprintf(), getchar() etc. Once you include "stdio.h" in your program, all these functions are available for use.

### User-defined functions

As mentioned earlier, C allow programmers to define functions. Such functions created by the user are called user-defined functions.

Depending upon the complexity and requirement of the program, you can create as many user-defined functions as you want.

### How user-defined function works?

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

```
void functionName()
```

```
{  
    .....  
    .....  
}
```

```
int main()
```

```
{  
    .....  
    .....  
}
```

```
functionName();
```

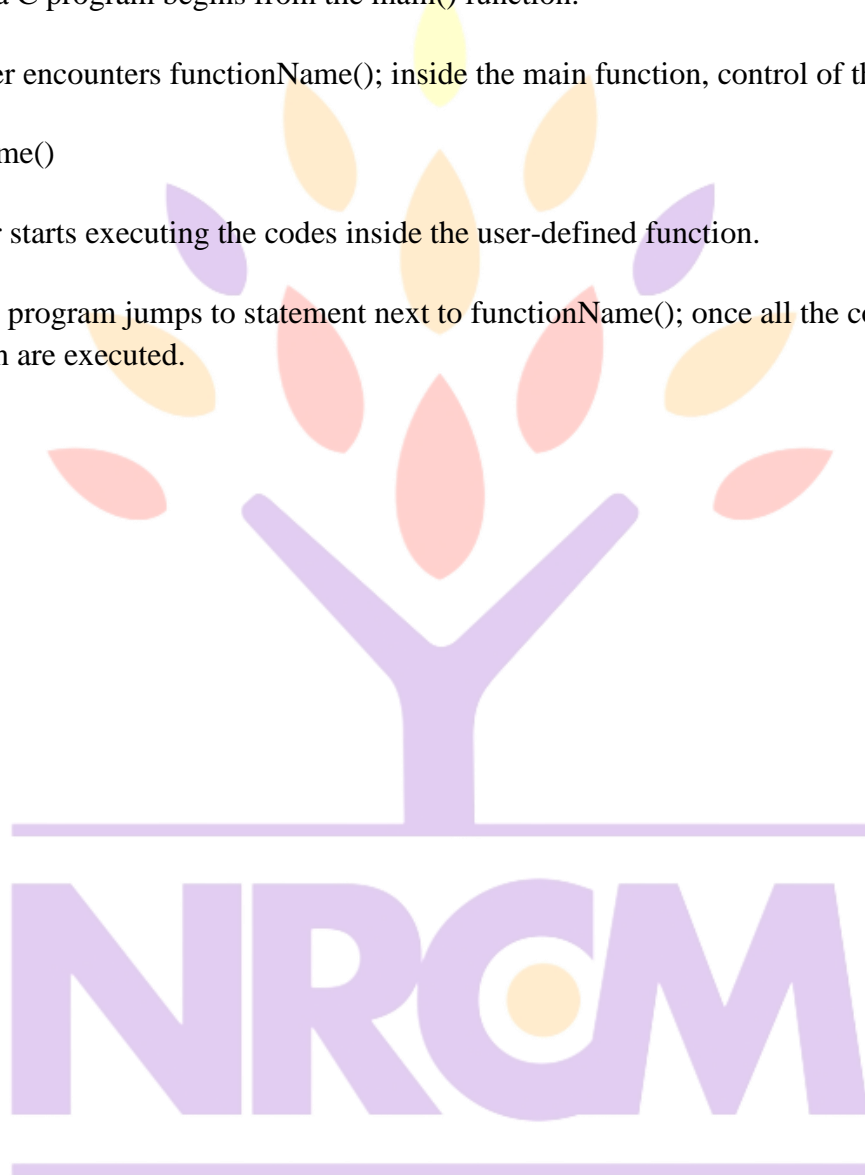
```
... ..  
... ..  
}
```

The execution of a C program begins from the main() function.

When the compiler encounters functionName(); inside the main function, control of the program jumps to  
void functionName()

And, the compiler starts executing the codes inside the user-defined function.

The control of the program jumps to statement next to functionName(); once all the codes inside the function definition are executed.



your roots to success...



## How function works in C programming?

```
#include <stdio.h>

void functionName()
{
    ... ..
    ... ..
}

int main()
{
    ... ..
    ... ..
    functionName();
}

... ..
... ..
```

Remember, function name is an identifier and should be unique.

This is just an overview on user-defined function. Visit these pages to learn more on:

- [User-defined Function in C programming](#)
- [Types of user-defined Functions](#)

### Advantages of user-defined function

1. The program will be easier to understand, maintain and debug.
2. Reusable codes that can be used in other programs
3. A large program can be divided into smaller modules. Hence, a large project can be divided among many programmers.

## C Programming User-defined functions

You will learn to create user-defined functions in C programming in this article.

A function is a block of code that performs a specific task.

C allows you to define functions according to your need. These functions are known as user-defined functions. For example:

Suppose, you need to create a circle and color it depending upon the radius and color. You can create two functions to solve this problem:

- createCircle() function
- color() function

### Example: User-defined function

Here is an example to add two integers. To perform this task, a user-defined function addNumbers() is defined.

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

```
int addNumbers(int a, int b);           // function prototype
```

```
int main(){
```

```
    int n1,n2,sum;
```

```
    printf("Enters two numbers: ");
```

```
    scanf("%d %d",&n1,&n2);
```

```
    sum = addNumbers(n1, n2);          // function call
```

```
    printf("sum = %d",sum);
```

```
    return 0;}
```

```
int addNumbers(int a,int b)           // function definition    {
```

```
    int result;
```

```
    result = a+b;
```

```
return result;           // return statement}
```

## Function prototype

A function prototype is simply the declaration of a function that specifies function's name, parameters and return type. It doesn't contain function body.

A function prototype gives information to the compiler that the function may later be used in the program.

## Syntax of function prototype

```
returnType functionName(type1 argument1, type2 argument2,...);
```

In the above example, `int addNumbers(int a, int b);` is the function prototype which provides following information to the compiler:

1. name of the function is `addNumbers()`
2. return type of the function is `int`
3. two arguments of type `int` are passed to the function

The function prototype is not needed if the user-defined function is defined before the `main()` function.

## Calling a function

Control of the program is transferred to the user-defined function by calling it.

## Syntax of function call

```
functionName(argument1, argument2, ...);
```

In the above example, function call is made using `addNumbers(n1,n2);` statement inside the `main()`.

## Function definition

Function definition contains the block of code to perform a specific task i.e. in this case, adding two numbers and returning it.

## Syntax of function definition

```
returnType functionName(type1 argument1, type2 argument2, ...)
```

```
{  
    //body of the function  
}
```

When a function is called, the control of the program is transferred to the function definition. And, the compiler starts executing the codes inside the body of a function.

## Passing arguments to a function

In programming, argument refers to the variable passed to the function. In the above example, two variables *n1* and *n2* are passed during function call.

The parameters *a* and *b* accepts the passed arguments in the function definition. These arguments are called formal parameters of the function.

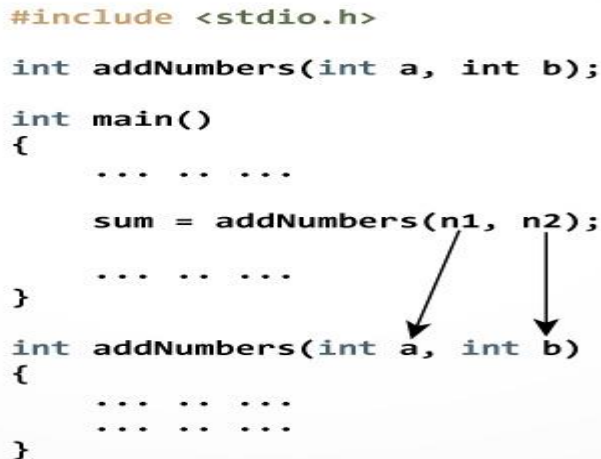
### How to pass arguments to a function?

```
#include <stdio.h>

int addNumbers(int a, int b);

int main()
{
    ... ..
    sum = addNumbers(n1, n2);
    ... ..
}

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

A diagram illustrating argument passing. In the code, the function call `addNumbers(n1, n2);` is in the `main()` function. Two arrows originate from `n1` and `n2` and point to the parameters `a` and `b` in the function definition `int addNumbers(int a, int b)`.

The type of arguments passed to a function and the formal parameters must match, otherwise the compiler throws error.

If *n1* is of char type, *a* also should be of char type. If *n2* is of float type, variable *b* also should be of float type. A function can also be called without passing an argument.

## Return Statement

The return statement terminates the execution of a function and returns a value to the calling function. The program control is transferred to the calling function after return statement.

In the above example, the value of variable *result* is returned to the variable *sum* in the `main()` function.

your roots to success...

## Return statement of a Function

```
#include <stdio.h>

int addNumbers(int a, int b);

int main()
{
    ... .. ...

    sum = addNumbers(n1, n2);

    ... .. ...
}

int addNumbers(int a, int b)
{
    ... .. ...
    return result;
}
```

sum = result

### Syntax of return statement

return (expression);

For example,

return a;

return (a+b);

The type of value returned from the function and the return type specified in function prototype and function definition must match.

### Types of User-defined Functions in C Programming

For better understanding of arguments and return value from the function, user-defined functions can be categorized as:

- 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.

The 4 programs below check whether an integer entered by the user is a prime number or not. And, all these programs generate the same output.

### Example #1: No arguments passed and no return Value

```
#include <stdio.h>

void checkPrimeNumber();

int main(){
    checkPrimeNumber();    // no argument is passed to prime()

    return 0;}

// return type of the function is void because no value is returned from the function
void checkPrimeNumber(){
    int n, i, flag=0;

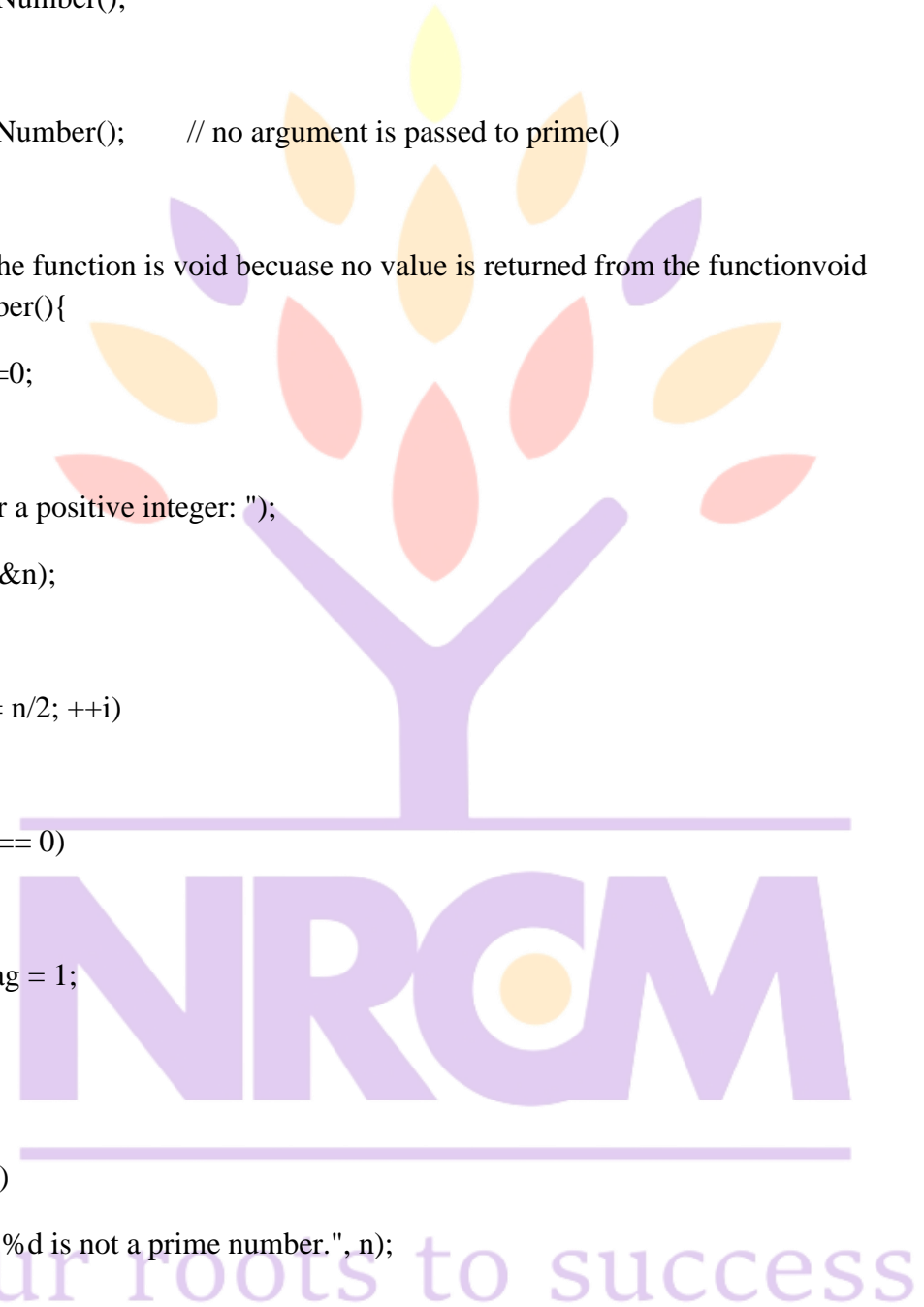
    printf("Enter a positive integer: ");
    scanf("%d",&n);

    for(i=2; i <= n/2; ++i)
    {
        if(n%i == 0)
        {
            flag = 1;
        }
    }

    if (flag == 1)

        printf("%d is not a prime number.", n);
    else

        printf("%d is a prime number.", n);}


```

The checkPrimeNumber() function takes input from the user, checks whether it is a prime number or not and displays it on the screen.

The empty parentheses in checkPrimeNumber(); statement inside the main() function indicates that no argument is passed to the 聽 function.

The return type of the function is void. Hence, no value is returned from the function.

### Example #2: No arguments passed but a return value

```
#include <stdio.h>int getInteger();

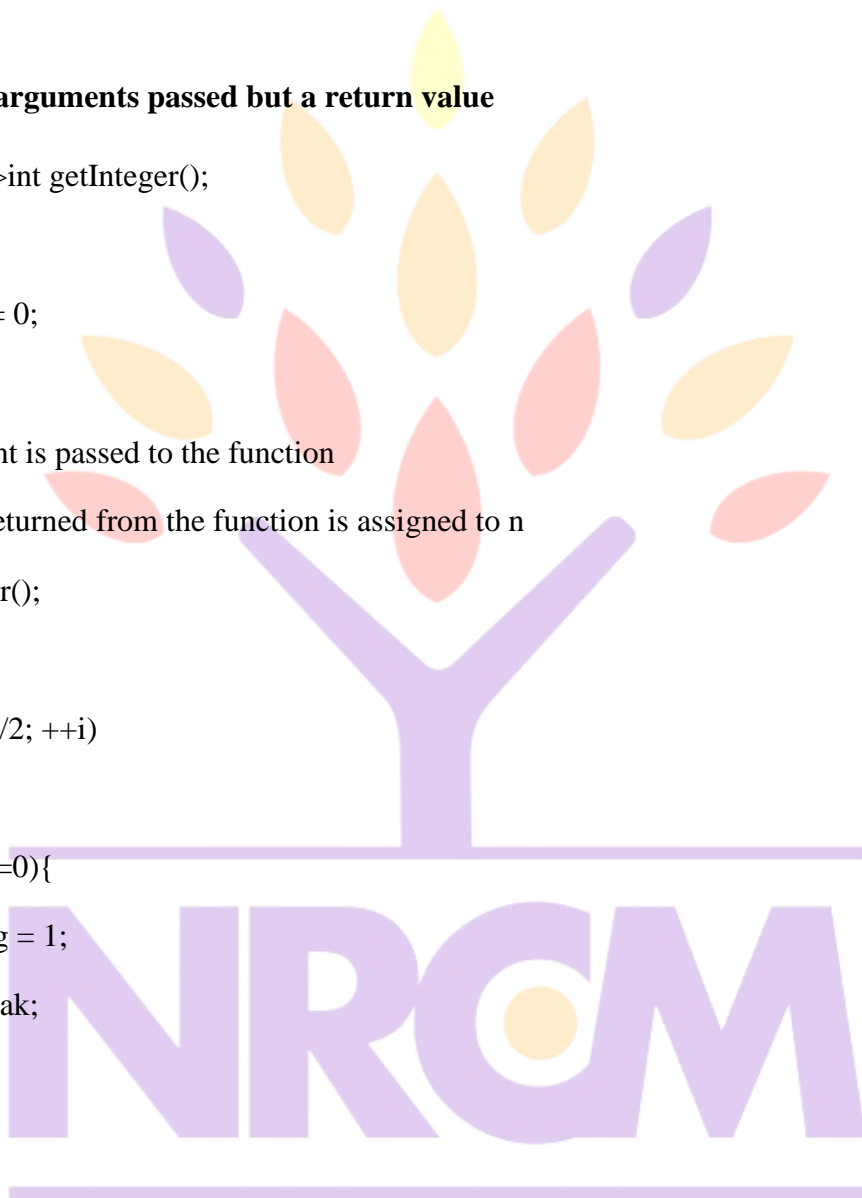
int main(){
    int n, i, flag = 0;

    // no argument is passed to the function
    // the value returned from the function is assigned to n
    n = getInteger();

    for(i=2; i<=n/2; ++i)
    {
        if(n%i==0){
            flag = 1;
            break;
        }
    }

    if (flag == 1)
        printf("%d is not a prime number.", n);
    else
        printf("%d is a prime number.", n);

    return 0;}
```



your roots to success...

```
// getInteger() function returns integer entered by the user
int getInteger(){
    int n;

    printf("Enter a positive integer: ");
    scanf("%d",&n);

    return n;}

```

The empty parentheses in `n = getInteger();` statement indicates that no argument is passed to the function. And, the value returned from the function is assigned to *n*.

Here, the `getInteger()` function takes input from the user and returns it. The code to check whether a number is prime or not is inside the `main()` function.

### Example #3: Argument passed but no return value

```
#include <stdio.h>
void checkPrimeAndDisplay(int n);

int main(){
    int n;

    printf("Enter a positive integer: ");
    scanf("%d",&n);

    // n is passed to the function
    checkPrimeAndDisplay(n);

    return 0;}

// void indicates that no value is returned from the function
void checkPrimeAndDisplay(int n){
    int i, flag = 0;

    for(i=2; i <= n/2; ++i)
    {
        if(n%i == 0){

```



```

        flag = 1;
        break;
    }
}
if(flag == 1)
    printf("%d is not a prime number.",n);
else
    printf("%d is a prime number.", n);}

```

The integer value entered by the user is passed to checkPrimeAndDisplay() function.

Here, the checkPrimeAndDisplay() function checks whether the argument passed is a prime number or not and displays the appropriate message.

#### Example #4: Argument passed and a return value

```

#include <stdio.h>int checkPrimeNumber(int n);
int main(){
    int n, flag;

    printf("Enter a positive integer: ");
    scanf("%d",&n);

    // n is passed to the checkPrimeNumber() function
    // the value returned from the function is assigned to flag variable
    flag = checkPrimeNumber(n);

    if(flag==1)
        printf("%d is not a prime number",n);
    else
        printf("%d is a prime number",n);
}

```

```

return 0;}

// integer is returned from the function
int checkPrimeNumber(int n){
    /* Integer value is returned from function checkPrimeNumber() */
    int i;

    for(i=2; i <= n/2; ++i)
    {
        if(n%i == 0)
            return 1;
    }

    return 0;}

```

The input from the user is passed to checkPrimeNumber() function. The checkPrimeNumber() function checks whether the passed argument is prime or not. If the passed argument is a prime number, the function returns 0. If the passed argument is a non-prime number, the function returns 1. The return value is assigned to *flag* variable.

Then, the appropriate message is displayed from the main() function.

### Which approach is better?

Well, it depends on the problem you are trying to solve. In case of this problem, the last approach is better.

A function should perform a specific task. The checkPrimeNumber() function doesn't take input from the user nor it displays the appropriate message. It only checks whether a number is prime or not, which makes code modular, easy to understand and debug.

### C Programming Recursion

A function that calls itself is known as a recursive function. And, this technique is known as recursion.

#### How recursion works?

```

void recurse()
{
    ... ..
    recurse();
}

```

```
.....
}

int main()
{
.....
recurse();
.....
}
```

### How does recursion work?

```
void recurse()
{
.....
recurse();
.....
}

int main()
{
.....
recurse();
.....
}
```

The diagram illustrates the flow of a recursive call. A line from the `recurse();` statement in the `main()` function points to the `recurse()` function definition. From the `recurse();` statement inside the `recurse()` function, a line points back to the `recurse()` function definition, forming a loop. This loop is labeled "recursive call".

The recursion continues until some condition is met to prevent it.

To prevent infinite recursion, if...else statement (or similar approach) can be used where one branch makes the recursive call and other doesn't.

### Example: Sum of Natural Numbers Using Recursion

```
#include <stdio.h>int sum(int n);

int main(){
    int number, result;
```

```
printf("Enter a positive integer: ");  
scanf("%d", &number);  
  
result = sum(number);  
  
printf("sum=%d", result);}  
int sum(int num){  
    if (num!=0)  
        return num + sum(num-1); // sum() function calls itself  
    else  
        return num;}
```

### Output

Enter a positive integer:

3

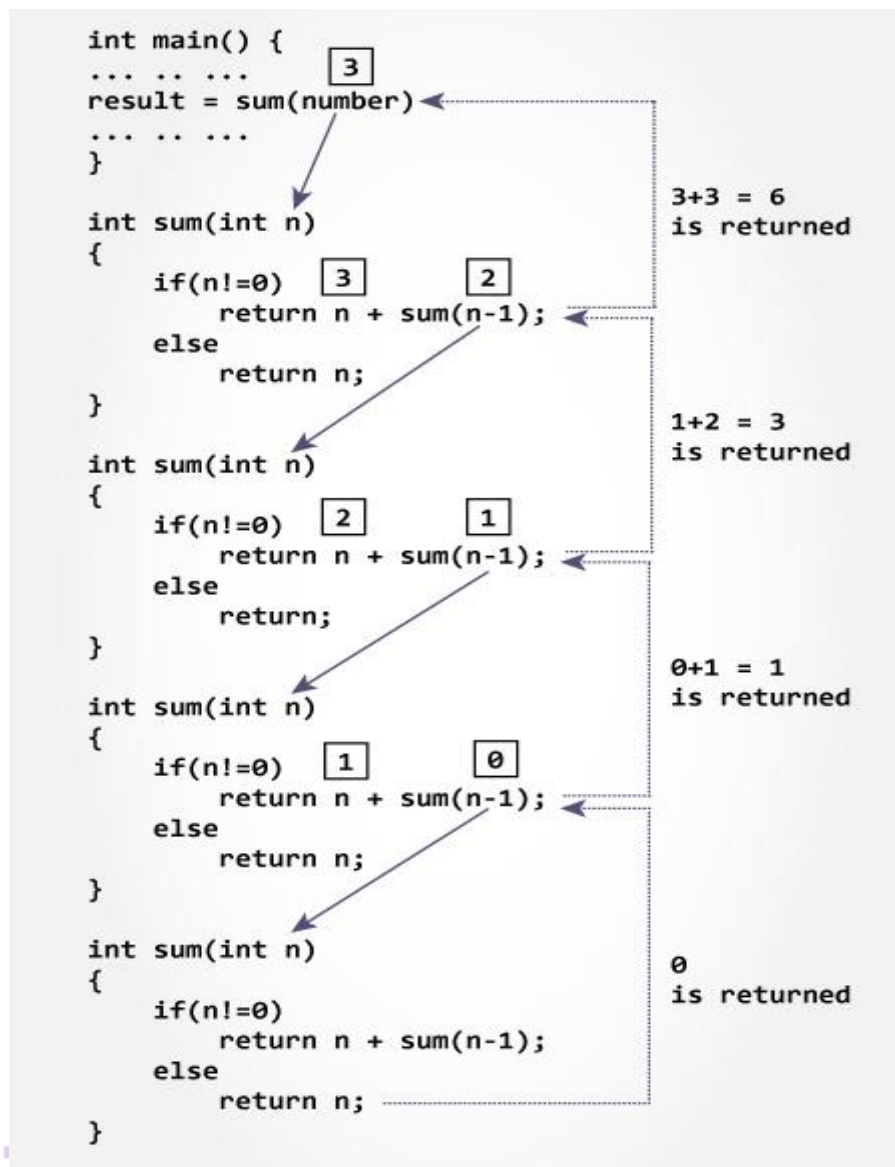
6

Initially, the sum() is called from the main() function with *number* passed as an argument.

Suppose, the value of *num* is 3 initially. During next function call, 2 is passed to the sum() function. This process continues until *num* is equal to 0.

When *num* is equal to 0, the if condition fails and the else part is executed returning the sum of integers to the main() function.

your roots to success...



## Advantages and Disadvantages of Recursion

Recursion makes program elegant and cleaner. All algorithms can be defined recursively which makes it easier to visualize and prove. 聽

If the speed of the program is vital then, you should avoid using recursion. Recursions use more memory and are generally slow. Instead, you can use loop.

## Scope and Lifetime of a variable.

Every variable in C programming has two properties: type and storage class.

Type refers to the data type of a variable. And, storage class determines the scope and lifetime of a variable.

There are 4 types of storage class:

1. automatic
2. external

3. static
4. register

### Local Variable

The variables declared inside the function are automatic or local variables.

The local variables exist only inside the function in which it is declared. When the function exits, the local variables are destroyed.

```
int main() {
    int n; // n is a local variable to main() function
    ... ..
}
void func() {
    int n1; // n1 is local to func() function
}
```

In the above code, *n1* is destroyed when `func()` exits. Likewise, *n* gets destroyed when `main()` exits.

### Global Variable

Variables that are declared outside of all functions are known as external variables. External or global variables are accessible to any function.

#### Example #1: External Variable

```
#include <stdio.h> void display();
int n = 5; // global variable
int main(){
    ++n; // variable n is not declared in the main() function
    display();
    return 0;}
void display(){
    ++n; // variable n is not declared in the display() function
    printf("n = %d", n);}
```

#### Output

n = 7

Suppose, a global variable is declared in file1. If you try to use that variable in a different file file2, the compiler will complain. To solve this problem, keyword `extern` is used in file2 to indicate that the external variable is declared in another file.

### Register Variable

The register keyword is used to declare register variables. Register variables were supposed to be faster than local variables.

However, modern compilers are very good at code optimization and there is a rare chance that using register variables will make your program faster.聽

Unless you are working on embedded system where you know how to optimize code for the given application, there is no use of register variables.

## Static Variable

A static variable is declared by using keyword static. For example;

```
static int i;
```

The value of a static variable persists until the end of the program.

### Example #2: Static Variable

```
#include <stdio.h>void display();
```

```
int main(){
```

```
    display();
```

```
    display();}void display(){
```

```
    static int c = 0;
```

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

```
    c += 5;}
```

### Output

```
0 5
```

During the first function call, the value of  $c$  is equal to 0. Then, it's value is increased by 5.

During the second function call, variable  $c$  is not initialized to 0 again. It's because  $c$  is a static variable. So, 5 is displayed on the screen.

## C Programming Arrays

An array is a collection of data that holds fixed number of values of same type. For example: if you want to store marks of 100 students, you can create an array for it.

```
float marks[100];
```

The size and type of arrays cannot be changed after its declaration.

Arrays are of two types:

1. One-dimensional arrays
2. Multidimensional arrays (will be discussed in next chapter)

## How to declare an array in C?

```
data_type array_name[array_size];
```

### For example,

```
float mark[5];
```

Here, we declared an array, *mark*, of floating-point type and size 5. Meaning, it can hold 5 floating-point values.

### Elements of an Array and How to access them?

You can access elements of an array by indices.

Suppose you declared an array *mark* as above. The first element is *mark[0]*, second element is *mark[1]* and so on.

```
mark[0] mark[1] mark[2] mark[3] mark[4]
```

--	--	--	--	--

### Few key notes:

- Arrays have 0 as the first index not 1. In this example, *mark[0]*
- If the size of an array is *n*, to access the last element, (*n-1*) index is used. In this example, *mark[4]*
- Suppose the starting address of *mark[0]* is 2120d. Then, the next address, *a[1]*, will be 2124d, address of *a[2]* will be 2128d and so on. It's because the size of a float is 4 bytes.

### How to initialize an array in C programming?

It's possible to initialize an array during declaration. For example,

```
int mark[5] = {19, 10, 8, 17, 9};
```

Another method to initialize array during declaration:

```
int mark[] = {19, 10, 8, 17, 9};
```

```
mark[0] mark[1] mark[2] mark[3] mark[4]
```

19	10	8	17	9
----	----	---	----	---



Here,

mark[0] is equal to 19

mark[1] is equal to 10

mark[2] is equal to 8

mark[3] is equal to 17

mark[4] is equal to 9

### How to insert and print array elements?

```
int mark[5] = {19, 10, 8, 17, 9}
```

```
// insert different value to third element
```

```
mark[3] = 9;
```

```
// take input from the user and insert in third elementscanf("%d", &mark[2]);
```

```
// take input from the user and insert in (i+1)th element
```

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

```
// print first element of an array
```

```
printf("%d", mark[0]);
```

```
// print ith element of an array
```

```
printf("%d", mark[i-1]);
```

### Example: C Arrays

```
// Program to find the average of n (n < 10) numbers using arrays
```

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

```
    int marks[10], i, n, sum = 0, average;
```

```
    printf("Enter n: ");
```

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

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

```
    {
```

```
        printf("Enter number%d: ",i+1);
```

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

```
        sum += marks[i];
```

```
}  
average = sum/n;  
printf("Average marks = %d", average);  
return 0;  
}
```

## Output

```
Enter n: 5  
Enter number1: 45  
Enter number2: 35  
Enter number3: 38  
Enter number4: 31  
Enter number5: 49  
Average = 39
```

## Important thing to remember when working with C arrays

Suppose you declared an array of 10 elements. Let's say,

```
int testArray[10];
```

You can use the array members from testArray[0] to testArray[9].

If you try to access array elements outside of its bound, let's say testArray[12], the compiler may not show any error. However, this may cause unexpected output (undefined behavior).

Before going further, checkout these array articles:

## C Programming Multidimensional Arrays

In this section, you will learn to work with multidimensional arrays (two dimensional and three dimensional array). In C programming, you can create array of an array known as multidimensional array. For example,

```
float x[3][4];
```

Here,  $x$  is a two-dimensional (2d) array. The array can hold 12 elements. You can think the array as table with 3 row and each row has 4 column.

	Column 1	Column 2	Column 3	Column 4
Row 1	x[0][0]	x[0][1]	x[0][2]	x[0][3]
Row 2	x[1][0]	x[1][1]	x[1][2]	x[1][3]
Row 3	x[2][0]	x[2][1]	x[2][2]	x[2][3]

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

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

Here, The array y can hold 24 elements.

You can think this example as: Each 2 elements have 4 elements, which makes 8 elements and each 8 elements can have 3 elements. Hence, the total number of elements is 24.

### How to initialize a multidimensional array?

There is more than one way to initialize a multidimensional array.

#### Initialization of a two dimensional array

```
// Different ways to initialize two dimensional array
```

```
int c[2][3] = {{1, 3, 0}, {-1, 5, 9}};
```

```
int c[][3] = {{1, 3, 0}, {-1, 5, 9}};
```

```
int c[2][3] = {1, 3, 0, -1, 5, 9};
```

Above code are three different ways to initialize a two dimensional arrays.

#### Initialization of a three dimensional 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} }  
};
```

### Example #1: Two Dimensional Array to store and display values

// C program to store temperature of two cities for a week and display it.

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

```
const int CITY = 2;const int WEEK = 7;
```

```
int main(){
```

```
    int temperature[CITY][WEEK];
```

```
    for (int i = 0; i < CITY; ++i) {
```

```
        for(int j = 0; j < WEEK; ++j) {
```

```
            printf("City %d, Day %d: ", i+1, j+1);
```

```
            scanf("%d", &temperature[i][j]);
```

```
        }
```

```
    }
```

```
    printf("\nDisplaying values: \n\n");
```

```
    for (int i = 0; i < CITY; ++i) {
```

```
        for(int j = 0; j < WEEK; ++j)
```

```
        {
```

```
            printf("City %d, Day %d = %d\n", i+1, j+1, temperature[i][j]);
```

```
        }
```

```
    }
```

```
    return 0;}
```

### Output

City 1, Day 1: 33

City 1, Day 2: 34

City 1, Day 3: 35

City 1, Day 4: 33

City 1, Day 5: 32

City 1, Day 6: 31

City 1, Day 7: 30

City 2, Day 1: 23

City 2, Day 2: 22

City 2, Day 3: 21

City 2, Day 4: 24

City 2, Day 5: 22

City 2, Day 6: 25

City 2, Day 7: 26

Displaying values:

City 1, Day 1 = 33

City 1, Day 2 = 34

City 1, Day 3 = 35

City 1, Day 4 = 33

City 1, Day 5 = 32

City 1, Day 6 = 31

City 1, Day 7 = 30

City 2, Day 1 = 23

City 2, Day 2 = 22

City 2, Day 3 = 21

City 2, Day 4 = 24

City 2, Day 5 = 22

City 2, Day 6 = 25

City 2, Day 7 = 26



**NRCM**

your roots to success...

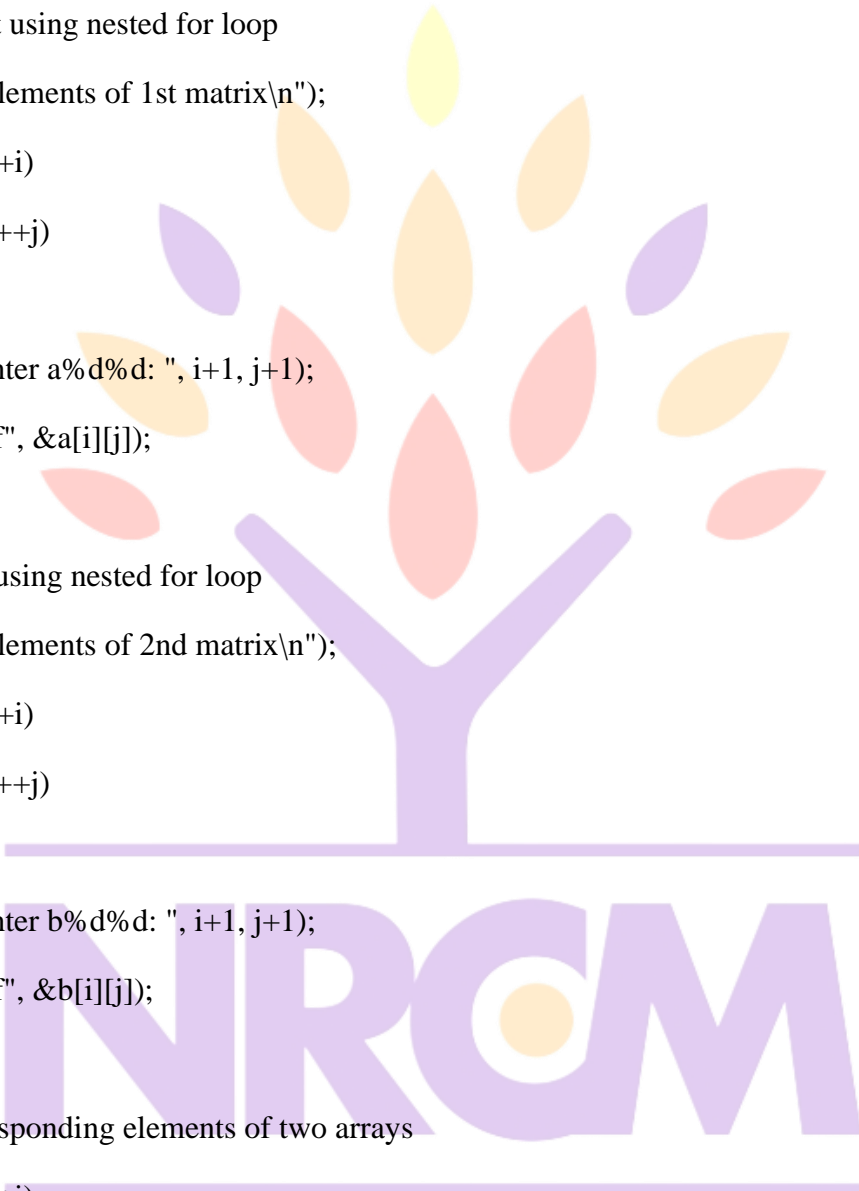
## Example #2: Sum of two matrices using Two dimensional arrays

C program to find the sum of two matrices of order 2\*2 using multidimensional arrays.

```
#include <stdio.h>int main(){
    float a[2][2], b[2][2], c[2][2];

    int i, j;

    // Taking input using nested for loop
    printf("Enter elements of 1st matrix\n");
    for(i=0; i<2; ++i)
        for(j=0; j<2; ++j)
        {
            printf("Enter a%d%d: ", i+1, j+1);
            scanf("%f", &a[i][j]);
        }
    // Taking input using nested for loop
    printf("Enter elements of 2nd matrix\n");
    for(i=0; i<2; ++i)
        for(j=0; j<2; ++j)
        {
            printf("Enter b%d%d: ", i+1, j+1);
            scanf("%f", &b[i][j]);
        }
    // adding corresponding elements of two arrays
    for(i=0; i<2; ++i)
        for(j=0; j<2; ++j)
        {
            c[i][j] = a[i][j] + b[i][j];
        }
    // Displaying the sum
    printf("\nSum Of Matrix:");
```



your roots to success...

```

for(i=0; i<2; ++i)
  for(j=0; j<2; ++j)
  {
    printf("%.1f\t", c[i][j]);

    if(j==1)
      printf("\n");
  }return 0;}

```

### Output

Enter elements of 1st matrix

Enter a11: 2;

Enter a12: 0.5;

Enter a21: -1.1;

Enter a22: 2;

Enter elements of 2nd matrix

Enter b11: 0.2;

Enter b12: 0;

Enter b21: 0.23;

Enter b22: 23;

Sum Of Matrix:

2.2      0.5

-0.9     25.0



### Example 3: Three Dimensional Array

**C Program to store values entered by the user in a three-dimensional array and display it.**

```

#include <stdio.h>int main(){
    // this array can store 12 elements

    int i, j, k, test[2][3][2];

    printf("Enter 12 values: \n");

    for(i = 0; i < 2; ++i) {

        for (j = 0; j < 3; ++j) {

```

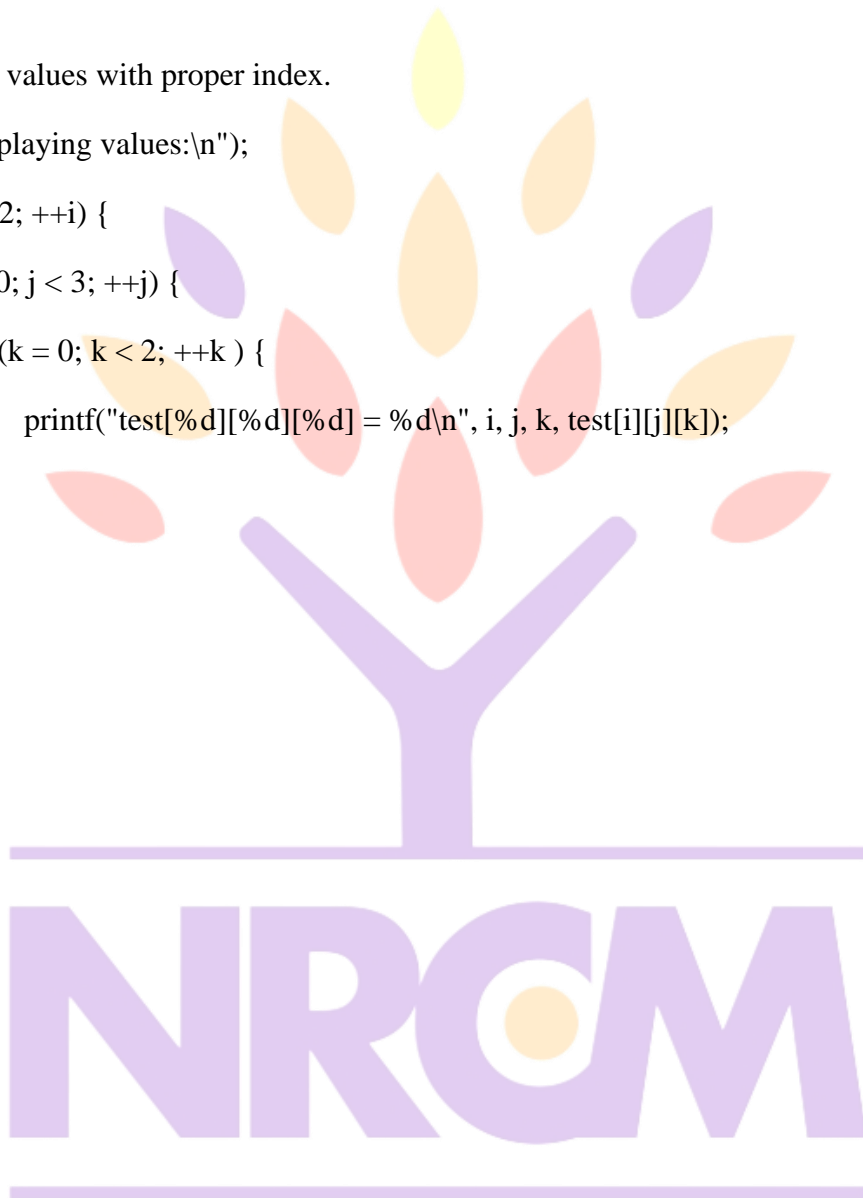
```
        for(k = 0; k < 2; ++k ) {
            scanf("%d", &test[i][j][k]);
        }
    }
}
// Displaying values with proper index.
printf("\nDisplaying values:\n");
for(i = 0; i < 2; ++i) {
    for (j = 0; j < 3; ++j) {
        for(k = 0; k < 2; ++k ) {
            printf("test[%d][%d][%d] = %d\n", i, j, k, test[i][j][k]);
        }
    }
}

return 0;}
```

### Output

Enter 12 values:

```
1
2
3
4
5
6
7 your roots to success...
8
9
10
11
```





12

Displaying Values:

```
test[0][0][0] = 1
test[0][0][1] = 2
test[0][1][0] = 3
test[0][1][1] = 4
test[0][2][0] = 5
test[0][2][1] = 6
test[1][0][0] = 7
test[1][0][1] = 8
test[1][1][0] = 9
test[1][1][1] = 10
test[1][2][0] = 11
test[1][2][1] = 12
```

### How to pass arrays to a function in C Programming?

In C programming, a single array element or an entire array can be passed to a function.

This can be done for both one-dimensional array or a multi-dimensional array.

#### Passing One-dimensional Array In Function

Single element of an array can be passed in similar manner as passing variable to a function.

#### C program to pass a single element of an array to function

```
#include <stdio.h>
void display(int age){
    printf("%d", age);}
int main(){
    int ageArray[] = { 2, 3, 4 };
    display(ageArray[2]); //Passing array element ageArray[2] only.
    return 0;}
```

#### Output

### Passing an entire one-dimensional array to a function

While passing arrays as arguments to the function, only the name of the array is passed (,i.e, starting address of memory area is passed as argument).

**C program to pass an array containing age of person to a function. This function should find average age and display the average age in main function.**

```
#include <stdio.h>float average(float age[]);

int main(){
    float avg, age[] = { 23.4, 55, 22.6, 3, 40.5, 18 };
    avg = average(age); /* Only name of array is passed as argument. */
    printf("Average age=%.2f", avg);
    return 0;}

float average(float age[]){
    int i;
    float avg, sum = 0.0;
    for (i = 0; i < 6; ++i) {
        sum += age[i];
    }
    avg = (sum / 6);
    return avg;}
```

### Output

Average age=27.08

your roots to success...

### Passing Multi-dimensional Arrays to Function

To pass two-dimensional array to a function as an argument, starting address of memory area reserved is passed as in one dimensional array

**#Example: Pass two-dimensional arrays to a function**

```
#include <stdio.h>void displayNumbers(int num[2][2]);int main(){  
  
    int num[2][2], i, j;  
    printf("Enter 4 numbers:\n");  
    for (i = 0; i < 2; ++i)  
        for (j = 0; j < 2; ++j)  
            scanf("%d", &num[i][j]);  
  
    // passing multi-dimensional array to displayNumbers function  
    displayNumbers(num);  
    return 0;}  
  
void displayNumbers(int num[2][2]){  
    // Instead of the above line,  
    // void displayNumbers(int num[][2]) is also valid  
    int i, j;  
    printf("Displaying:\n");  
    for (i = 0; i < 2; ++i)  
        for (j = 0; j < 2; ++j)  
            printf("%d\n", num[i][j]);}
```

### Output

Enter 4 numbers:

2

3

4

5

Displaying:

2

3

4

5



your roots to success...

## C Programming Pointers and Arrays

Arrays are closely related to pointers in C programming but the important difference between them is that, a pointer variable takes different addresses as value whereas, in case of array it is fixed.

This can be demonstrated by an example:

```
#include <stdio.h>int main(){  
    char charArr[4];  
    int i;  
  
    for(i = 0; i < 4; ++i)  
    {  
        printf("Address of charArr[%d] = %u\n", i, &charArr[i]);  
    }  
  
    return 0;}
```

When you run the program, the output will be:

```
Address of charArr[0] = 28ff44  
Address of charArr[1] = 28ff45  
Address of charArr[2] = 28ff46  
Address of charArr[3] = 28ff47
```

**Note:** You may get different address of an array.

Notice, that there is an equal difference (difference of 1 byte) between any two consecutive elements of array *charArr*.

But, since pointers just point at the location of another variable, it can store any address.

### Relation between Arrays and Pointers

Consider an array:

```
int arr[4];
```

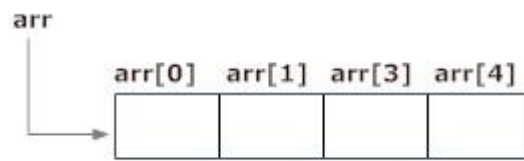


Figure: Array as Pointer

In C programming, name of the array always points to address of the first element of an array.

In the above example,  $arr$  and  $\&arr[0]$  points to the address of the first element.

$\&arr[0]$  is equivalent to  $arr$

Since, the addresses of both are the same, the values of  $arr$  and  $\&arr[0]$  are also the same.

$arr[0]$  is equivalent to  $*arr$  (value of an address of the pointer)

Similarly,

$\&arr[1]$  is equivalent to  $(arr + 1)$  AND,  $arr[1]$  is equivalent to  $*(arr + 1)$ .

$\&arr[2]$  is equivalent to  $(arr + 2)$  AND,  $arr[2]$  is equivalent to  $*(arr + 2)$ .

$\&arr[3]$  is equivalent to  $(arr + 3)$  AND,  $arr[3]$  is equivalent to  $*(arr + 3)$ .

.

.

$\&arr[i]$  is equivalent to  $(arr + i)$  AND,  $arr[i]$  is equivalent to  $*(arr + i)$ .

In C, you can declare an array and can use pointer to alter the data of an array.

**Example: Program to find the sum of six numbers with arrays and pointers**

```
#include <stdio.h>int main(){
    int i, classes[6],sum = 0;
    printf("Enter 6 numbers:\n");
    for(i = 0; i < 6; ++i)
    {
        // (classes + i) is equivalent to &classes[i]
        scanf("%d", (classes + i));
```

```
// *(classes + i) is equivalent to classes[i]
sum += *(classes + i);
}
printf("Sum = %d", sum);
return 0;}
```

### Output

Enter 6 numbers:

2

3

4

5

3

4

Sum = 21

### Call by Reference: Using pointers [With Examples]

When a pointer is passed as an argument to a function, address of the memory location is passed instead of the value.

This is because, pointer stores the location of the memory, and not the value.

### Example of Pointer And Functions

#### Program to swap two number using call by reference.

```
/* C Program to swap two numbers using pointers and function. */#include <stdio.h>void swap(int *n1, int *n2);
```

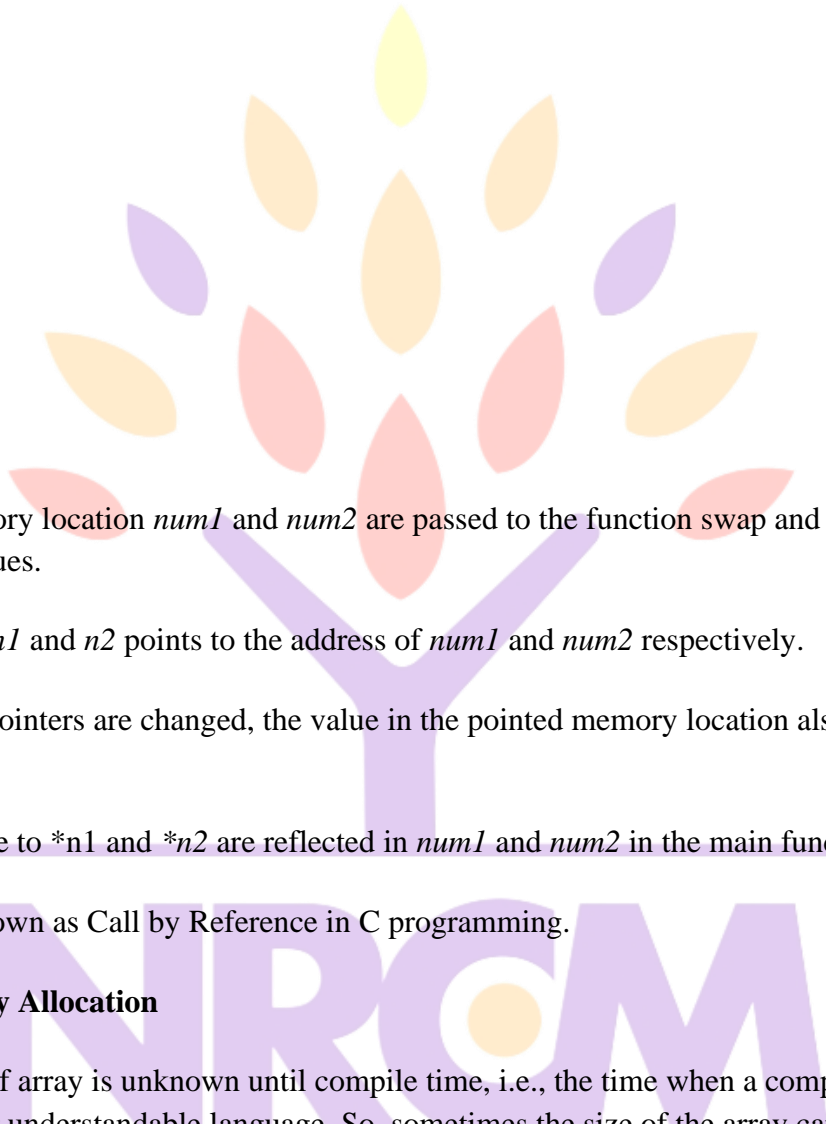
```
int main(){
```

```
int num1 = 5, num2 = 10;
```

```
// address of num1 and num2 is passed to the swap function
```

```
swap( &num1, &num2);
```

```
printf("Number1 = %d\n", num1);
```

```
printf("Number2 = %d", num2);  
return 0;}  
  
void swap(int * n1, int * n2){  
    // pointer n1 and n2 points to the address of num1 and num2 respectively  
    int temp;  
    temp = *n1;  
    *n1 = *n2;  
    *n2 = temp;}  

```

### Output

Number1 = 10

Number2 = 5

The address of memory location *num1* and *num2* are passed to the function *swap* and the pointers *\*n1* and *\*n2* accept those values.

So, now the pointer *n1* and *n2* points to the address of *num1* and *num2* respectively.

When, the value of pointers are changed, the value in the pointed memory location also changes correspondingly.

Hence, changes made to *\*n1* and *\*n2* are reflected in *num1* and *num2* in the main function.

This technique is known as Call by Reference in C programming.

### C Dynamic Memory Allocation

In C, the exact size of array is unknown until compile time, i.e., the time when a compiler compiles your code into a computer understandable language. So, sometimes the size of the array can be insufficient or more than required.

Dynamic memory allocation allows your program to obtain more memory space while running, or to release it if it's not required.

In simple terms, Dynamic memory allocation allows you to manually handle memory space for your program.

Although, C language inherently does not have any technique to allocate memory dynamically, there are 4 library functions under "stdlib.h" for dynamic memory allocation.

Function	Use of Function
malloc()	Allocates requested size of bytes and returns a pointer first byte of allocated space
calloc()	Allocates space for an array elements, initializes to zero and then returns a pointer to memory
free()	deallocate the previously allocated space
realloc()	Change the size of previously allocated space

### C malloc()

The name malloc stands for "memory allocation".

The function malloc() reserves a block of memory of specified size and return a pointer of type void which can be casted into pointer of any form.

#### Syntax of malloc()

```
ptr = (cast-type*) malloc(byte-size)
```

Here, *ptr* is pointer of cast-type. The malloc() function returns a pointer to an area of memory with size of byte size. If the space is insufficient, allocation fails and returns NULL pointer.

```
ptr = (int*) malloc(100 * sizeof(int));
```

This statement will allocate either 200 or 400 according to size of int 2 or 4 bytes respectively and the pointer points to the address of first byte of memory.

### C calloc()

The name calloc stands for "contiguous allocation".

The only difference between malloc() and calloc() is that, malloc() allocates single block of memory whereas calloc() allocates multiple blocks of memory each of same size and sets all bytes to zero.

#### Syntax of calloc()

```
ptr = (cast-type*)calloc(n, element-size);
```

This statement will allocate contiguous space in memory for an array of *n* elements. For example:

```
ptr = (float*) calloc(25, sizeof(float));
```



This statement allocates contiguous space in memory for an array of 25 elements each of size of float, i.e, 4 bytes.

## C free()

Dynamically allocated memory created with either calloc() or malloc() doesn't get freed on its own. You must explicitly use free() to release the space.

### syntax of free()

```
free(ptr);
```

This statement frees the space allocated in the memory pointed by ptr.

### Example #1: Using C malloc() and free()

**Write a C program to find sum of n elements entered by user. To perform this program, allocate memory dynamically using malloc() function.**

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

```
int main(){
```

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

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

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

```
    ptr = (int*) malloc(num * sizeof(int)); //memory allocated using malloc
```

```
    if(ptr == NULL)
```

```
    {
```

```
        printf("Error! memory not allocated.");
```

```
        exit(0);
```

```
    }
```

```
    printf("Enter elements of array: ");
```

```
    for(i = 0; i < num; ++i)
```

```
    {
```

```
scanf("%d", ptr + i);  
  
sum += *(ptr + i);  
  
}
```

```
printf("Sum = %d", sum);  
  
free(ptr);  
  
return 0;}
```

### Example #2: Using C calloc() and free()

Write a C program to find sum of n elements entered by user. To perform this program, allocate memory dynamically using calloc() function.

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

```
int main(){
```

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

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

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

```
ptr = (int*) calloc(num, sizeof(int));
```

```
if(ptr == NULL)
```

```
{
```

```
printf("Error! memory not allocated.");
```

```
exit(0);
```

```
}
```

```
printf("Enter elements of array: ");
```

```
for(i = 0; i < num; ++i)
```

```
{
```

```
scanf("%d", ptr + i);
```

```
sum += *(ptr + i);
```

```
}

printf("Sum = %d", sum);

free(ptr);

return 0;}
```

### C realloc()

If the previously allocated memory is insufficient or more than required, you can change the previously allocated memory size using `realloc()`.

#### Syntax of realloc()

```
ptr = realloc(ptr, newsize);
```

Here, *ptr* is reallocated with size of *newsize*.

#### Example #3: Using realloc()

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

```
int main(){
```

```
    int *ptr, i , n1, n2;
```

```
    printf("Enter size of array: ");
```

```
    scanf("%d", &n1);
```

```
    ptr = (int*) malloc(n1 * sizeof(int));
```

```
    printf("Address of previously allocated memory: ");
```

```
    for(i = 0; i < n1; ++i)
```

```
        printf("%u\t", ptr + i);
```

```
    printf("\nEnter new size of array: ");
```

```
    scanf("%d", &n2);
```

```
    ptr = realloc(ptr, n2);
```

```
    for(i = 0; i < n2; ++i)
```

```
        printf("%u\t", ptr + i);
```

```
    return 0;}
```

### C Programming Structure

Structure is a collection of variables of different types under a single name.

**For example:** You want to store some information about a person: his/her name, citizenship number and salary. You can easily create different variables *name*, *citNo*, *salary* to store these information separately.

However, in the future, you would want to store information about multiple persons. Now, you'd need to create different variables for each information per person: *name1*, *citNo1*, *salary1*, *name2*, *citNo2*, *salary2*

You can easily visualize how big and messy the code would look. Also, since no relation between the variables (information) would exist, it's going to be a daunting task.

A better approach will be to have a collection of all related information under a single name Person, and use it for every person. Now, the code looks much cleaner, readable and efficient as well.

This collection of all related information under a single name Person is a structure.

### Structure Definition in C

Keyword struct is used for creating a structure.

#### Syntax of structure

```
struct structure_name
{
    data_type member1;
    data_type member2;
    .
    .
    data_type memeber;
};
```

**Note:** Don't forget the semicolon };聽 in the ending line.

We can create the structure for a person as mentioned above as:

```
struct person
{
    char name[50];
    int citNo;
    float salary;
};
```

This declaration above creates the derived data type struct person.

#### Structure variable declaration

When a structure is defined, it creates a user-defined type but, no storage or memory is allocated.

For the above structure of a person, variable can be declared as:

```
struct person
{
    char name[50];
    int citNo;
    float salary;
};
```

```
int main()
```

```
{
    struct person person1, person2, person3[20];
    return 0;
}
```

Another way of creating a structure variable is:

```
struct person
{
    char name[50];
    int citNo;
    float salary;
} person1, person2, person3[20];
```

In both cases, two variables *person1*, *person2* and an array *person3* having 20 elements of type **struct person** are created.

### Accessing members of a structure

There are two types of operators used for accessing members of a structure.

1. Member operator(.)
2. Structure pointer operator(->)

Any member of a structure can be accessed as:

```
structure_variable_name.member_name
```

Suppose, we want to access salary for variable *person2*. Then, it can be accessed as:

```
person2.salary
```

### Example of structure

**Write a C program to add two distances entered by user. Measurement of distance should be in inch and feet. (Note: 12 inches = 1 foot)**

```
#include <stdio.h>
struct Distance{
    int feet;
    float inch;}
dist1, dist2, sum;

int main(){
```

```
printf("1st distance\n");
```

```
// Input of feet for structure variable dist1
```

```
printf("Enter feet: ");
```

```
scanf("%d", &dist1.feet);
```

```
// Input of inch for structure variable dist1
```

```
printf("Enter inch: ");
```

```
scanf("%f", &dist1.inch);
```

```
printf("2nd distance\n");
```

```
// Input of feet for structure variable dist2
```

```
printf("Enter feet: ");
```

```
scanf("%d", &dist2.feet);
```

```
// Input of feet for structure variable dist2
```

```
printf("Enter inch: ");
```

```
scanf("%f", &dist2.inch);
```

```
sum.feet = dist1.feet + dist2.feet;
```

```
sum.inch = dist1.inch + dist2.inch;
```

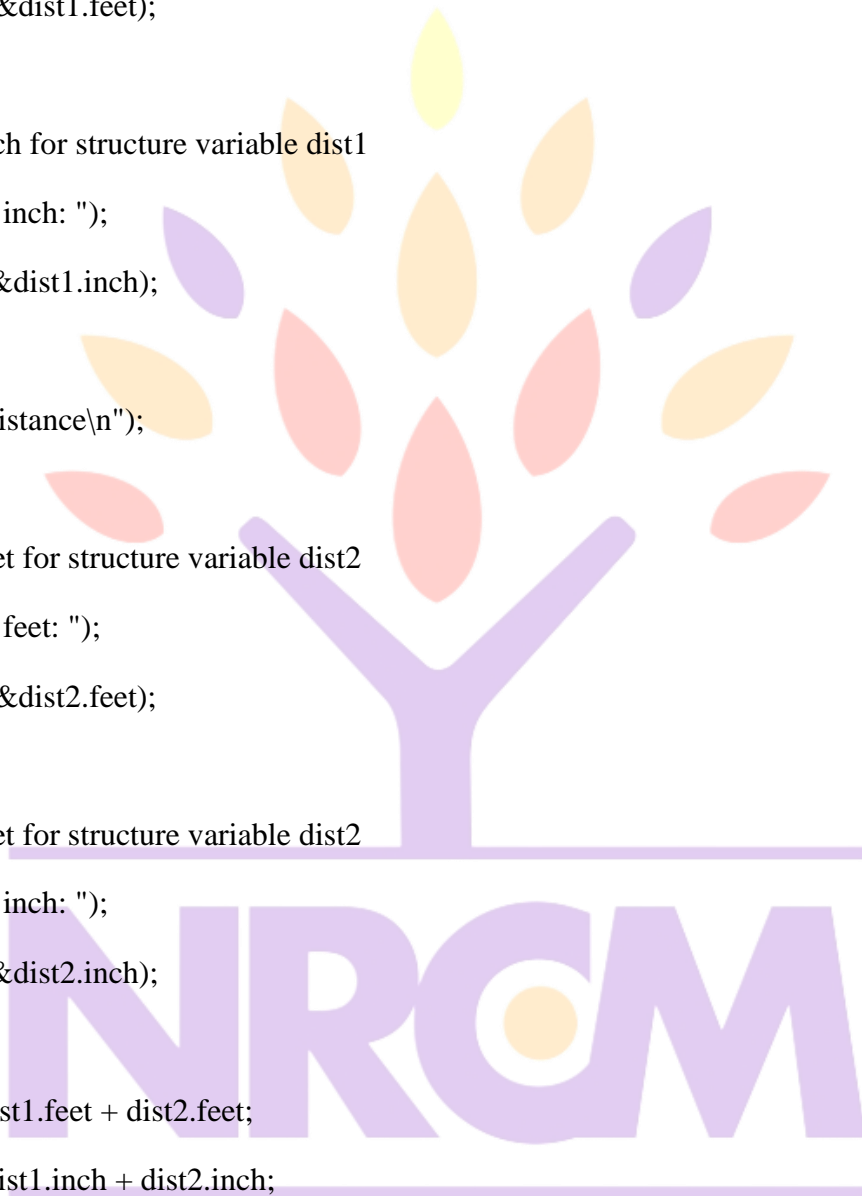
```
if (sum.inch > 12)
```

```
{
```

```
    //If inch is greater than 12, changing it to feet.
```

```
    ++sum.feet;
```

```
    sum.inch = sum.inch - 12;
```



your roots to success...

```
}

// printing sum of distance dist1 and dist2
printf("Sum of distances = %d\`-%.1f\`", sum.feet, sum.inch);
return 0;}
```

### Output

```
1st distance
Enter feet: 12
Enter inch: 7.9
2nd distance
Enter feet: 2
Enter inch: 9.8
Sum of distances = 15'-5.7"
```

### Keyword typedef while using structure

Writing struct structure\_name variable\_name; to declare a structure variable isn't intuitive as to what it signifies, and takes some considerable amount of development time.

So, developers generally use typedef to name the structure as a whole. For example:

```
typedef struct complex
```

```
{
    int imag;
    float real;
```

```
} comp;
```

```
int main()
```

```
{
    comp comp1, comp2;
}
```

Here, typedef keyword is used in creating a type *comp* (which is of type as struct complex).

Then, two structure variables *comp1* and *comp2* are created by this *comp* type.

### Structures within structures

Structures can be nested within other structures in C programming.

```
struct complex
{
    int imag_value;
    float real_value;
};

struct number
{
    struct complex comp;
    int real;
} num1, num2;
```

Suppose, you want to access *imag\_value* for *num2* structure variable then, following structure member is used.

num2.comp.imag\_value

### Passing structures to a function

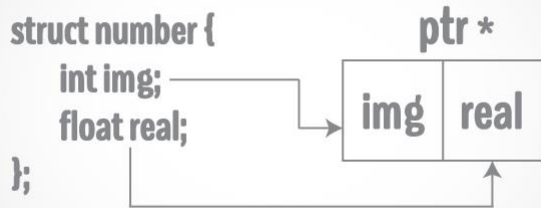
There are mainly two ways to pass structures to a function:

1. Passing by value
2. Passing by reference

### C Programming Structure and Pointer

your roots to success...





Structures can be created and accessed using pointers. A pointer variable of a structure can be created as below:

```

struct name {
    member1;
    member2;
    .
    .
};

```

```

int main(){
    struct name *ptr;
}

```

Here, the pointer variable of type **struct name** is created.

### Accessing structure's member through pointer

A structure's member can be accessed through pointer in two ways:

1. Referencing pointer to another address to access memory
2. Using dynamic memory allocation

### 1. Referencing pointer to another address to access the memory\_ Consider an example to access structure's member through pointer.

```

#include <stdio.h>typedef struct person{
    int age;
    float weight;};
int main(){

```

```

struct person *personPtr, person1;

personPtr = &person1;           // Referencing pointer to memory address of person1

printf("Enter integer: ");

scanf("%d",&(*personPtr).age);

printf("Enter number: ");

scanf("%f",&(*personPtr).weight);

printf("Displaying: ");

printf("%d%f",(*personPtr).age,(*personPtr).weight);

return 0;}

```

In this example, the pointer variable of type struct person is referenced to the address of *person1*. Then, only the structure member through pointer can be accessed.

### Using -> operator to access structure pointer member

Structure pointer member can also be accessed using -> operator.

(\*personPtr).age is same as personPtr->age

(\*personPtr).weight is same as personPtr->weight

### 2. Accessing structure member through pointer using dynamic memory allocation

To access structure member using pointers, memory can be allocated dynamically using malloc() function defined under "stdlib.h" header file.

#### Syntax to use malloc()

```
ptr = (cast-type*) malloc(byte-size)
```

#### Example to use structure's member through pointer using malloc() function.

```
#include <stdio.h>#include <stdlib.h>struct person {
```

```
int age;
```

```
float weight;
```

```
char name[30];};
```

```
int main(){
```

```
struct person *ptr;
```

```
int i, num;
```

```
printf("Enter number of persons: ");
```

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

```

ptr = (struct person*) malloc(num * sizeof(struct person));
// Above statement allocates the memory for n structures with pointer personPtr pointing to base address
for(i = 0; i < num; ++i)
{
    printf("Enter name, age and weight of the person respectively:\n");
    scanf("%s%d%f", &(ptr+i)->name, &(ptr+i)->age, &(ptr+i)->weight);
}
printf("Displaying Infromation:\n");
for(i = 0; i < num; ++i)
    printf("%s\t%d\t%.2f\n", (ptr+i)->name, (ptr+i)->age, (ptr+i)->weight);
return 0;}

```

### Output

Enter number of persons: 2

Enter name, age and weight of the person respectively:

Adam

2

3.2

Enter name, age and weight of the person respectively:

Eve

6

2.3

Displaying Information:

Adam 2 3.20

Eve 6 2.30

### How to pass structure to a function in C programming?

In this article, you'll find relevant examples to pass structures as an argument to a function, and use them in your program.

In C, structure can be passed to functions by two methods:

1. Passing by value (passing actual value as argument)
2. Passing by reference (passing address of an argument)

### Passing structure by value

A structure variable can be passed to the function as an argument as a normal variable.

If structure is passed by value, changes made to the structure variable inside the function definition does not reflect in the originally passed structure variable.

### C program to create a structure student, containing name and roll and display the information.

```
#include <stdio.h>struct student{  
  
    char name[50];  
  
    int roll;};  
  
void display(struct student stu);// function prototype should be below to the structure declaration otherwise  
compiler shows error  
  
int main(){  
  
    struct student stud;  
  
    printf("Enter student's name: ");  
  
    scanf("%s", &stud.name);  
  
    printf("Enter roll number:");  
  
    scanf("%d", &stud.roll);  
  
    display(stud);    // passing structure variable stud as argument  
  
    return 0;}void display(struct student stu){  
  
    printf("Output\nName: %s",stu.name);  
  
    printf("\nRoll: %d",stu.roll);}
```

### Output

Enter student's name: Kevin Amla

Enter roll number: 149

### Output

Name: Kevin Amla

Roll: 149

## Passing structure by reference

The memory address of a structure variable is passed to function while passing it by reference.

If structure is passed by reference, changes made to the structure variable inside function definition reflects in the originally passed structure variable.

### C program to add two distances (feet-inch system) and display the result without the return statement.

```
#include <stdio.h>struct distance{
    int feet;
    float inch;};void add(struct distance d1,struct distance d2, struct distance *d3);
int main(){
    struct distance dist1, dist2, dist3;
    printf("First distance\n");
    printf("Enter feet: ");
    scanf("%d", &dist1.feet);
    printf("Enter inch: ");
    scanf("%f", &dist1.inch);
    printf("Second distance\n");
    printf("Enter feet: ");
    scanf("%d", &dist2.feet);
    printf("Enter inch: ");
    scanf("%f", &dist2.inch);
    add(dist1, dist2, &dist3);

    //passing structure variables dist1 and dist2 by value whereas passing structure variable dist3 by
reference
    printf("\nSum of distances = %d\`-%.1f\`", dist3.feet, dist3.inch);

    return 0;}void add(struct distance d1,struct distance d2, struct distance *d3) {
    //Adding distances d1 and d2 and storing it in d3
    d3->feet = d1.feet + d2.feet;
    d3->inch = d1.inch + d2.inch;
```

```

if (d3->inch >= 12) {      /* if inch is greater or equal to 12, converting it to feet. */
    d3->inch -= 12;
    ++d3->feet;
}
}

```

### Output

First distance

Enter feet: 12

Enter inch: 6.8

Second distance

Enter feet: 5

Enter inch: 7.5

Sum of distances = 18'-2.3"

In this program, structure variables *dist1* and *dist2* are passed by value to the add function (because value of *dist1* and *dist2* does not need to be displayed in main function). But, *dist3* is passed by reference, i.e., address of *dist3* (&*dist3*) is passed as an argument.

Due to this, the structure pointer variable *d3* inside the add function points to the address of *dist3* from the calling main function. So, any change made to the *d3* variable is seen in *dist3* variable in main function.

As a result, the correct sum is displayed in the output.

### C Programming Unions

Unions are quite similar to structures in C. Like structures, unions are also derived types.

```

union car

```

```

{
    char name[50];

```

```

    int price;
};

```

Defining a union is as easy as replacing the keyword **struct** with the keyword **union**.

### How to create union variables?

Union variables can be created in similar manner as structure variables.

```
union car
{
    char name[50];
    int price;
} car1, car2, *car3;
```

**OR**

```
union car
{
    char name[50];
    int price;
};

int main()
{
    union car car1, car2, *car3;
    return 0;
}
```

In both cases, union variables *car1*, *car2* and union pointer variable *car3* of type **union car** is created.

### Accessing members of a union

Again, the member of unions can be accessed in similar manner as structures.

In the above example, suppose you want to access *price* for union variable *car1*, it can be accessed as:

```
car1.price
```

Likewise, if you want to access *price* for the union pointer variable *car3*, it can be accessed as:..

```
(*car3).price
```

or;

```
car3->price
```

### Difference between union and structure

Though unions are similar to structure in so many ways, the difference between them is crucial to understand.

The primary difference can be demonstrated by this example:

```
#include <stdio.h>
union unionJob{
    //defining a union
    char name[32];
    float salary;
    int workerNo;} uJob;
struct structJob{
    char name[32];
    float salary;
    int workerNo;} sJob;
int main(){
    printf("size of union = %d", sizeof(uJob));
    printf("\nsize of structure = %d", sizeof(sJob));
    return 0;}
```

**Output**

size of union = 32  
size of structure = 40

**More memory is allocated to structures than union**

As seen in the above example, there is a difference in memory allocation between union and structure.

The amount of memory required to store a structure variable is the sum of memory size of all members.



Fig: Memory allocation in case of structure

But, the memory required to store a union variable is the memory required for the largest element of an union.



name



32 bytes

Fig: Memory allocation in case of union

### Only one union member can be accessed at a time

In the case of structure, all of its members can be accessed at any time.

But, in the case of union, only one of its members can be accessed at a time and all other members will contain garbage values.

```
#include <stdio.h>union job{
    char name[32];
    float salary;
    int workerNo;} job1;
```

```
int main(){
    printf("Enter name:\n");
    scanf("%s", &job1.name);

    printf("Enter salary: \n");
    scanf("%f", &job1.salary);

    printf("Displaying\nName :%s\n", job1.name);
    printf("Salary: %.1f", job1.salary);
```

```
return 0;}
```

### Output

Enter name

Hillary

Enter salary

1234.23

Displaying

Name: f%Bary

Salary: 1234.2

**Note:** You may get different garbage value for the name.

Initially in the program, *Hillary* is stored in `job1.name` and all other members of `job1`, i.e. `salary`, `workerNo`, will contain garbage values.

But, when user enters the value of salary, 1234.23 will be stored in `job1.salary` and other members, i.e. `name`, `workerNo`, will now contain garbage values.

Thus in the output, *salary* is printed accurately but, *name* displays some random string.

### Passing Union To a Function

Union can be passed in similar manner as structures in C programming.

### C Programming Files I/O

There are a large number of functions to handle file I/O (Input Output) in C. In this tutorial, you will learn to handle standard I/O in C using `fprintf()`, `fscanf()`, `fread()`, `fwrite()`, `fseek()` and more.

In C programming, file is a place on your physical disk where information is stored.

### Why files are needed?

- When a program is terminated, the entire data is lost. Storing in a file will preserve your data even if the program terminates.
- If you have to enter a large number of data, it will take a lot of time to enter them all. However, if you have a file containing all the data, you can easily access the contents of the file using few commands in C.
- You can easily move your data from one computer to another without any changes.

### Types of Files

When dealing with files, there are two types of files you should know about:

1. Text files
2. Binary files

#### 1. Text files

Text files are the normal `.txt` files that you can easily create using Notepad or any simple text editors.

When you open those files, you'll see all the contents within the file as plain text. You can easily edit or delete the contents.

They take minimum effort to maintain, are easily readable, and provide least security and takes bigger storage space.

## 2. Binary files

Binary files are mostly the .bin files in your computer.

Instead of storing data in plain text, they store it in the binary form (0's and 1's).

They can hold higher amount of data, are not readable easily and provides a better security than text files.

### File Operations

In C, you can perform four major operations on the file, either text or binary:

1. Creating a new file
2. Opening an existing file
3. Closing a file
4. Reading from and writing information to a file

### Working with files

When working with files, you need to declare a pointer of type file. This declaration is needed for communication between the file and program.

```
FILE *fptr;
```

#### Opening a file - for creation and edit

Opening a file is performed using the library function in the "**stdio.h**" header file: `fopen()`.

The syntax for opening a file in standard I/O is:

```
ptr = fopen("filename", "mode")
```

For Example:

```
fopen("E:\\cprogram\\newprogram.txt", "w");
```

```
fopen("E:\\cprogram\\oldprogram.bin", "rb");
```

- Let's suppose the file `newprogram.txt` doesn't exist in the location `E:\cprogram`. The first function creates a new file named `newprogram.txt` and opens it for writing as per the mode `'w'`. The writing mode allows you to create and edit (overwrite) the contents of the file.
- Now let's suppose the second binary file `oldprogram.bin` exists in the location `E:\cprogram`. The second function opens the existing file for reading in binary mode `'rb'`. The reading mode only allows you to read the file, you cannot write into the file.

Opening Modes in Standard I/O

<b>File Mode</b>	<b>Meaning of Mode</b>	<b>During Inexistence of file</b>
r	Open for reading.	If the file does not exist, fopen() returns NULL.
rb	Open for reading in binary mode.	If the file does not exist, fopen() returns NULL.
w	Open for writing.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
wb	Open for writing in binary mode.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
a	Open for append. i.e, Data is added to end of file.	If the file does not exist, it will be created.
ab	Open for append in binary mode. i.e, Data is added to end of file.	If the file does not exist, it will be created.
r+	Open for both reading and writing.	If the file does not exist, fopen() returns NULL.
rb+	Open for both reading and writing in binary mode.	If the file does not exist, fopen() returns NULL.
w+	Open for both reading and writing.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
wb+	Open for both reading and writing in binary mode.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
a+	Open for both reading and appending.	If the file does not exist, it will be created.
ab+	Open for both reading and appending in binary mode.	If the file does not exist, it will be created.

### **Closing a File**

The file (both text and binary) should be closed after reading/writing.

Closing a file is performed using library function fclose().

fclose(fp); //fp is the file pointer associated with file to be closed.

### **Reading and writing to a text file**

For reading and writing to a text file, we use the functions fprintf() and fscanf().

They are just the file versions of printf() and scanf(). The only difference is that, fprintf and fscanf expects a pointer to the structure FILE.

### Writing to a text file

#### Example 1: Write to a text file using fprintf()

```
#include <stdio.h>int main(){
    int num;
    FILE *fptr;
    fptr = fopen("C:\\program.txt","w");

    if(fptr == NULL)
    {
        printf("Error!");
        exit(1);
    }

    printf("Enter num: ");
    scanf("%d",&num);

    fprintf(fptr,"%d",num);
    fclose(fptr);

    return 0;}
```

This program takes a number from user and stores in the file program.txt.

After you compile and run this program, you can see a text file program.txt created in C drive of your computer. When you open the file, you can see the integer you entered.

### Reading from a text file

#### Example 2: Read from a text file using fscanf()

```

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

    int num;

    FILE *fptr;

    if ((fptr = fopen("C:\\program.txt","r")) == NULL){

        printf("Error! opening file");

        // Program exits if the file pointer returns NULL.

        exit(1);

    }

    fscanf(fptr,"%d", &num);

    printf("Value of n=%d", num);

    fclose(fptr);

    return 0;}

```

This program reads the integer present in the program.txt file and prints it onto the screen.

If you successfully created the file from **Example 1**, running this program will get you the integer you entered.

Other functions like fgetchar(), fputc() etc. can be used in similar way.

### Reading and writing to a binary file

Functions fread() and fwrite() are used for reading from and writing to a file on the disk respectively in case of binary files.

#### Writing to a binary file

To write into a binary file, you need to use the function fwrite(). The function takes four arguments: Address of data to be written in disk, Size of data to be written in disk, number of such type of data and pointer to the file where you want to write.

```
fwrite(address_data,size_data,numbers_data,pointer_to_file);
```

### Example 3: Writing to a binary file using fwrite()

```
#include <stdio.h>

struct threeNum{
    int n1, n2, n3;};

int main(){

    int n;

    struct threeNum num;

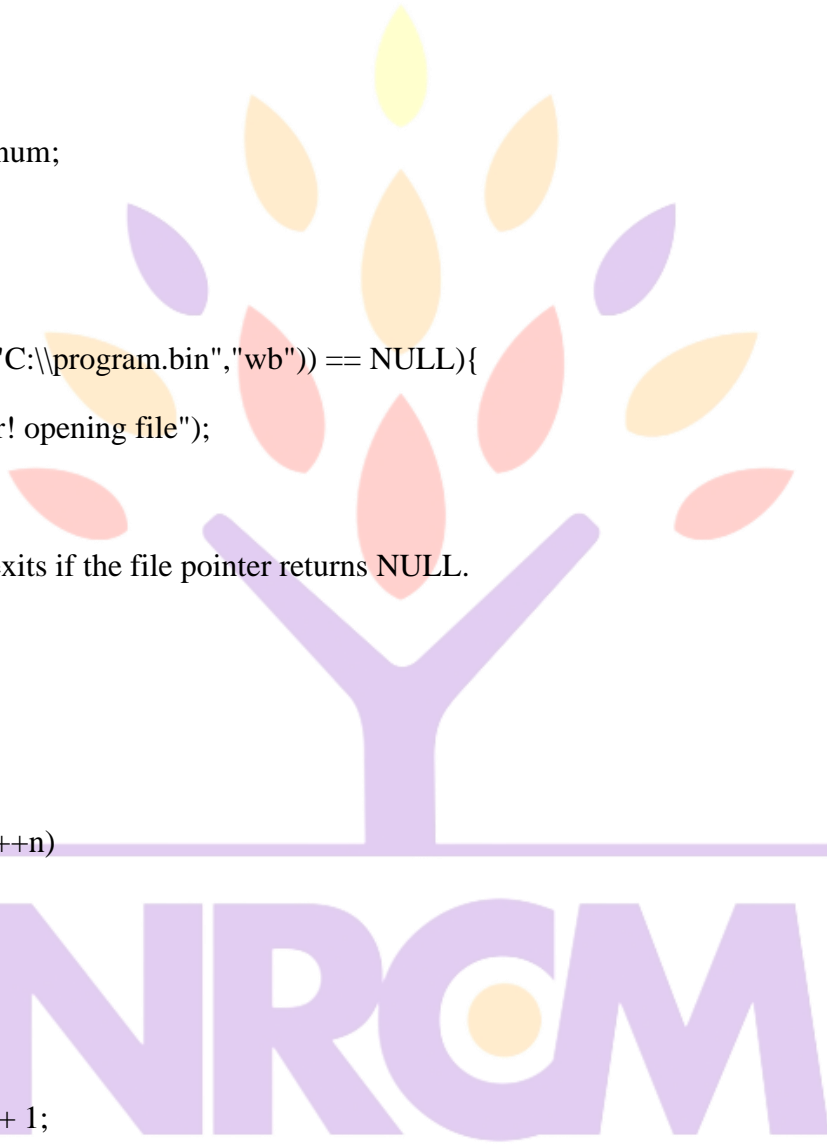
    FILE *fptr;

    if ((fptr = fopen("C:\\program.bin","wb")) == NULL){
        printf("Error! opening file");

        // Program exits if the file pointer returns NULL.
        exit(1);
    }

    for(n = 1; n < 5; ++n)
    {
        num.n1 = n;
        num.n2 = 5n;
        num.n3 = 5n + 1;
        fwrite(&num, sizeof(struct threeNum), 1, fptr);
    }
    fclose(fptr);

    return 0;}
```



your roots to success...

In this program, you create a new file program.bin in the C drive.

We declare a structure `threeNum` with three numbers - `n1`, `n2` and `n3`, and define it in the main function as `num`.

Now, inside the for loop, we store the value into the file using `fwrite`.

The first parameter takes the address of `num` and the second parameter takes the size of the structure `threeNum`.

Since, we're only inserting one instance of `num`, the third parameter is 1. And, the last parameter `*fptr` points to the file we're storing the data.

Finally, we close the file.

### Reading from a binary file

Function `fread()` also take 4 arguments similar to `fwrite()` function as above.

```
fread(address_data,size_data,numbers_data,pointer_to_file);
```

### Example 4: Reading from a binary file using `fread()`

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

```
struct threeNum{
```

```
    int n1, n2, n3;};
```

```
int main(){
```

```
    int n;
```

```
    struct threeNum num;
```

```
    FILE *fptr;
```

```
    if ((fptr = fopen("C:\\program.bin","rb")) == NULL){
```

```
        printf("Error! opening file");
```

```
        // Program exits if the file pointer returns NULL.
```

```
        exit(1);
```

```
    }
```

```
    for(n = 1; n < 5; ++n)
```



```

{
    fread(&num, sizeof(struct threeNum), 1, fptr);
    printf("n1: %d\n2: %d\n3: %d", num.n1, num.n2, num.n3);
}
fclose(fptr);

return 0;}

```

In this program, you read the same file program.bin and loop through the records one by one.

In simple terms, you read one threeNum record of threeNum size from the file pointed by *\*fptr* into the structure *num*.

You'll get the same records you inserted in Example 3.

### Getting data using fseek()

If you have many records inside a file and need to access a record at a specific position, you need to loop through all the records before it to get the record.

This will waste a lot of memory and operation time. An easier way to get to the required data can be achieved using fseek().

As the name suggests, fseek() seeks the cursor to the given record in the file.

### Syntax of fseek()

```
fseek(FILE * stream, long int offset, int whence)
```

The first parameter stream is the pointer to the file. The second parameter is the position of the record to be found, and the third parameter specifies the location where the offset starts.

### Different Whence in fseek

#### Whence

#### Meaning

SEKK\_SET Starts the offset from the beginning of the file.

SEKK\_END Starts the offset from the end of the file.

SEKK\_CUR Starts the offset from the current location of the cursor in the file.

### Example of fseek()

```

#include <stdio.h>

struct threeNum{
    int n1, n2, n3;};

int main(){

    int n;

    struct threeNum num;

    FILE *fptr;

    if ((fptr = fopen("C:\\program.bin","rb")) == NULL){
        printf("Error! opening file");

        // Program exits if the file pointer returns NULL.
        exit(1);
    }

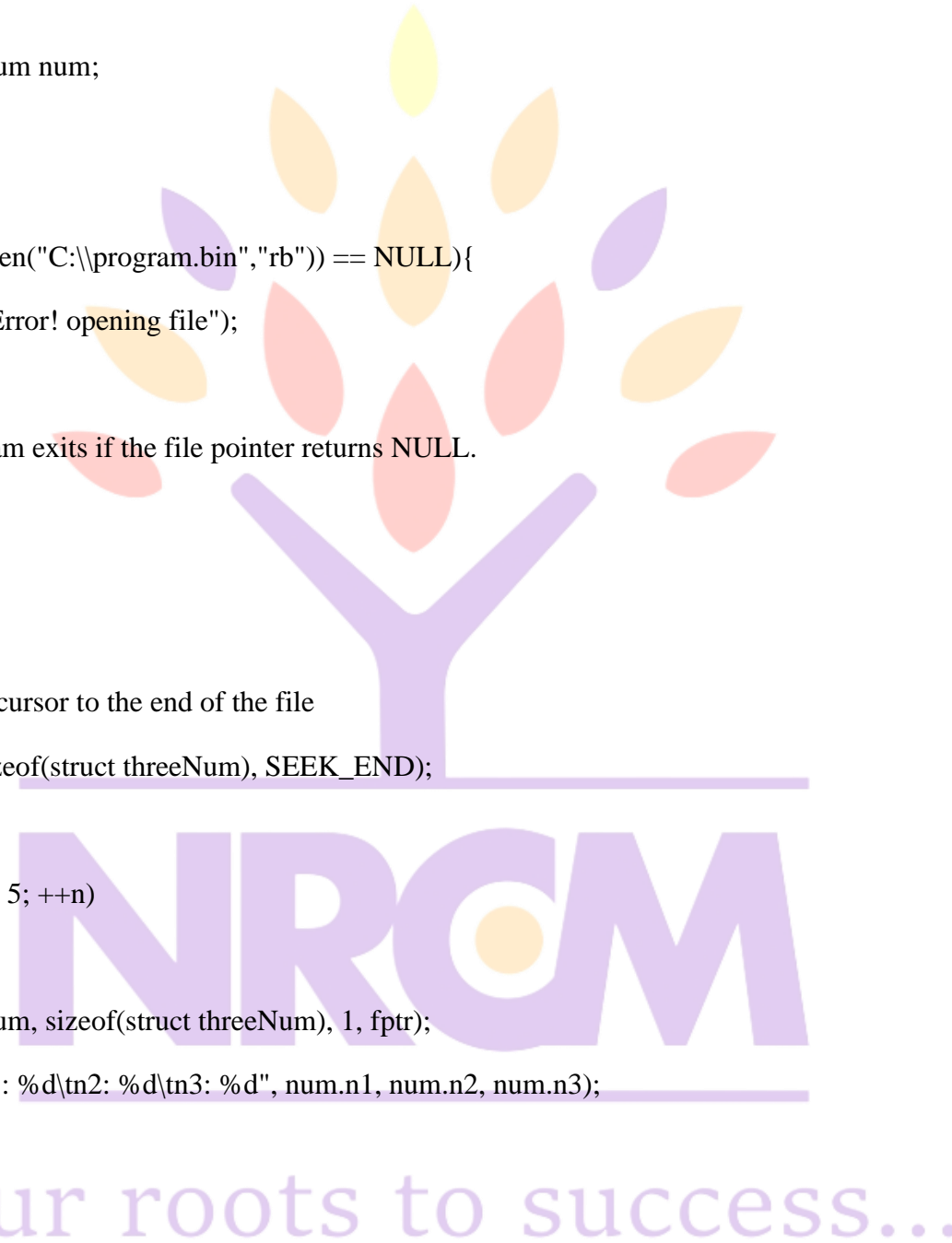
    // Moves the cursor to the end of the file
    fseek(fptr, sizeof(struct threeNum), SEEK_END);

    for(n = 1; n < 5; ++n)
    {
        fread(&num, sizeof(struct threeNum), 1, fptr);
        printf("n1: %d\t n2: %d\t n3: %d", num.n1, num.n2, num.n3);

    }
    fclose(fptr);

    return 0;}

```



This program will start reading the records from the file program.bin in the reverse order (last to first) and prints it.

## C Programming Enumeration

In this article, you will learn to work with enumeration (enum). Also, you will learn where enums are commonly used in C programming.

An enumeration is a user-defined data type that consists of integral constants. To define an enumeration, keyword enum is used.

```
enum flag { const1, const2, ..., constN };
```

Here, name of the enumeration is *flag*.

And, *const1, const2, ..., constN* are values of type *flag*.

By default, *const1* is 0, *const2* is 1 and so on. You can change default values of enum elements during declaration (if necessary).

```
// Changing default values of enum
```

```
enum suit {  
    club = 0,  
    diamonds = 10,  
    hearts = 20,  
    spades = 3,  
};
```

### Enumerated Type Declaration

When you create an enumerated type, only blueprint for the variable is created. Here's how you can create variables of enum type.

```
enum boolean { false, true };  
enum boolean check;
```

Here, a variable *check* of type enum boolean is created.

Here is another way to declare same *check* variable using different syntax.

```
enum boolean  
{  
    false, true  
} check;
```

## Example: Enumeration Type

```
#include <stdio.h>

enum week { sunday, monday, tuesday, wednesday, thursday, friday, saturday };

int main(){

    enum week today;

    today = wednesday;

    printf("Day %d",today+1);

    return 0;}
```

**Output:** Day 4

## Why enums are used in C programming?

Enum variable takes only one value out of many possible values. Example to demonstrate it,

```
#include <stdio.h>

enum suit {

    club = 0,

    diamonds = 10,

    hearts = 20,

    spades = 3 } card;

int main() {

    card = club;

    printf("Size of enum variable = %d bytes", sizeof(card));

    return 0;}
```

## Output

Size of enum variable = 4 bytes

It's because the size of an integer is 4 bytes.

This makes enum a good choice to work with flags.

You can accomplish the same task using structures. However, working with enums gives you efficiency along with flexibility.

## How to use enums for flags?

Let us take an example,

```
enum designFlags {  
    ITALICS = 1,  
    BOLD = 2,  
    UNDERLINE = 4} button;
```

Suppose you are designing a button for Windows application. You can set flags *ITALICS*, *BOLD* and *UNDERLINE* to work with text.

There is a reason why all the integral constants are power of 2 in above pseudocode.

// In binary

```
ITALICS = 00000001
```

```
BOLD = 00000010
```

```
UNDERLINE = 00000100
```

Since, the integral constants are power of 2, you can combine two or more flags at once without overlapping using bitwise OR | operator. This allows you to choose two or more flags at once. For example,

Example program:

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

```
enum designFlags {
```

```
    BOLD = 1,
```

```
    ITALICS = 2,
```

```
    UNDERLINE = 4};
```

```
int main() {
```

```
    int myDesign = BOLD | UNDERLINE;
```

```
    //    00000001
```

```
    // | 00000100
```

```
// _____  
// 00000101  
  
printf("%d", myDesign);  
  
return 0;}
```

Output

5

When the output is 5, you always know that bold and underline is used.

Also, you can add flag to your requirements.

```
if (myDesign & ITALICS) {  
    // code for italics  
}
```

Here, we have added italics to our design. Note, only code for italics is written inside if statement.

You can accomplish almost anything in C programming without using enumerations. However, they can be pretty handy in certain situations. That's what differentiates good programmers from great programmers.

### String operations (string.h)

language recognizes that strings are terminated by null character and is a different class of array by letting us input and output the array as a unit. To array out many of the string manipulations, C library supports a large number of string handling functions that can be used such as:

1. Length (number of characters in the string).
2. Concatenation (adding two or more strings)
3. Comparing two strings.
4. Substring (Extract substring from a given string)
5. Copy(copies one string over another)

#### strlen():

The strlen() function calculates the length of a given string.

```
//calculates the length of string before null character.
```

Example: C strlen() function

```
#include <stdio.h>
#include <string.h>
int main()
{
    char a[20]="Program";
    char b[20]={'P','r','o','g','r','a','m','\0'};
    char c[20];
    printf("Enter string: ");
    gets(c);
    printf("Length of string a = %d \n",strlen(a));
    //calculates the length of string before null character.
    printf("Length of string b = %d \n",strlen(b));
    printf("Length of string c = %d \n",strlen(c));
    return 0;
}
```

### Output

```
Enter string: String
Length of string a = 7
Length of string b = 7
Length of string c = 6
```

### 2.strcpy():

The strcpy() function copies the string pointed by source (including the null character) to the character array destination.

This function returns character array destination.

The strcpy() function is defined in string.h header file.

Example: C strcpy()

```

#include <stdio.h>
#include <string.h>
int main()
{
    char str1[10]= "awesome";
    char str2[10];
    char str3[10];
    strcpy(str2, str1);
    strcpy(str3, "well");
    puts(str2);
    puts(str3);
    return 0;
}

```

### Output

```

awesome
well

```

```

/* strcpy example */

```

```

#include <stdio.h>

```

```

#include <string.h>

```

```

int main ()

```

```

{

```

```

    char str1[]= "To be or not to be";

```

```

    char str2[40];

```

```

    char str3[40];

```

```

    strcpy ( str2, str1);

```

```

    /* partial copy (only 5 chars): */

```



your roots to success...



```
strncpy ( str3, str2, 5 );  
  
puts (str1);  
  
puts (str2);  
  
puts (str3);  
  
return 0;  
  
}
```

Output:

To be or not to be  
To be or not to be  
To be

**strcat():**

The function strcat() concatenates two strings.

In C programming, strcat() concatenates (joins) two strings.

The strcat() function is defined in <string.h> header file

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

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

```
int main()
```

```
{
```

```
    char str1[] = "hello i am ", str2[] = "sarfaraz";
```

```
    //concatenates str1 and str2 and resultant string is stored in str1.
```

```
    strcat(str1,str2);//str1=str1+str2;
```

```
    puts(str1);
```

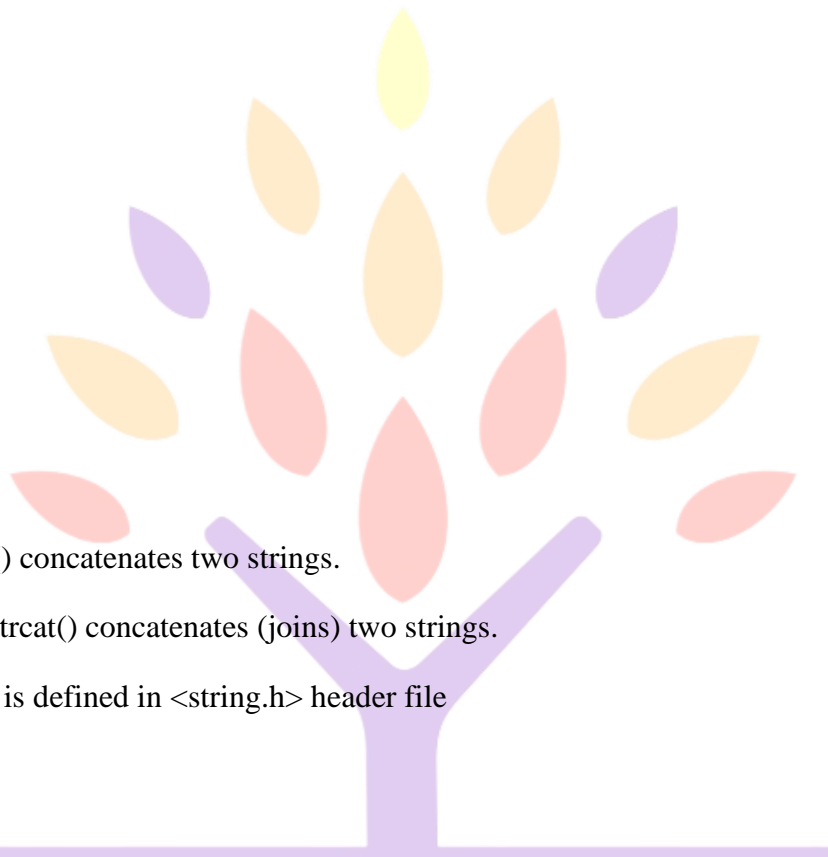
```
    puts(str2);
```

```
    return 0;
```

```
}
```

**Output**

hello i am sarfaraz



**NRCM**

your roots to success...

sarfaraz

```
/* strncat example */
```

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

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

```
int main ()
```

```
{
```

```
char str1[20];
```

```
char str2[20];
```

```
strcpy (str1,"hello");
```

```
strcpy (str2,"good afternoon");
```

```
strncat (str1, str2, 7);
```

```
puts (str1);
```

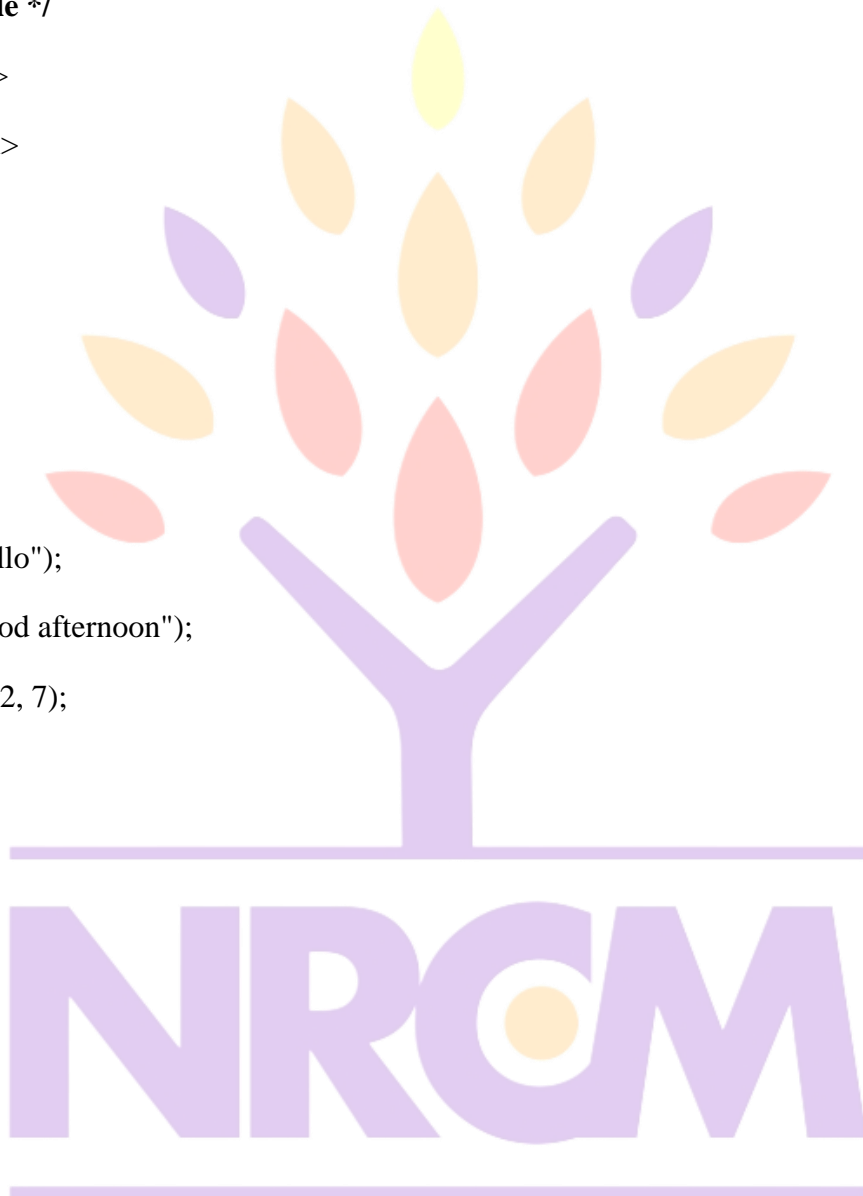
```
return 0;
```

```
}
```

Edit & Run

Output:

hellogood af



your roots to success...

**strlwr():**

strlwr( ) function converts a given string into lowercase.

Syntax for strlwr( ) function is given below.

```
#include<stdio.h>
#include<string.h>
int main()
{
    char str[ ] = "MODIFY This String To LOwer";
    printf("%s\n",strlwr (str));
    return 0;
}
```

**Output:**

modify this string to lower

**strupr( )** function converts a given string into uppercase.

Syntax forstrupr ( ) function is given below.

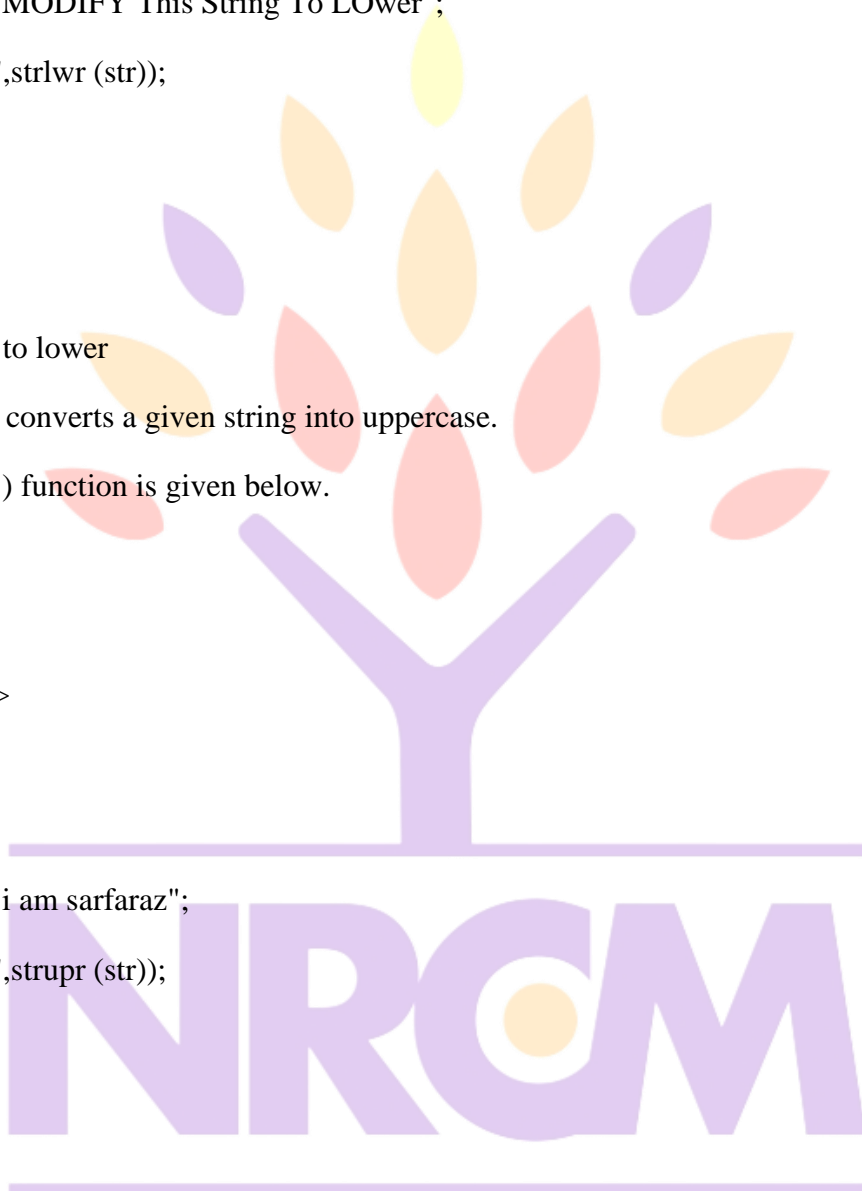
```
#include<stdio.h>
#include<string.h>
int main()
{
    char str[ ] = "i am sarfaraz";
    printf("%s\n",strupr (str));
    return 0;
}
```

**Output:**

I AM SARFARAZ

**Strrev():** reverse the given string

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



```

char name[30] = "Hello";

printf("String before strrev( ) : %s\n",name);

printf("String after strrev( ) : %s",strrev(name));

return 0;
}

```

**Output:**

String before strrev( ) : Hello

String after strrev( ) : olleH

**strcmp( )** function in C compares two given strings and returns zero if they are same.

•If length of string1 < string2, it returns < 0 value. If length of string1 > string2, it returns > 0 value. Syntax for strcmp( ) function is given below.

```

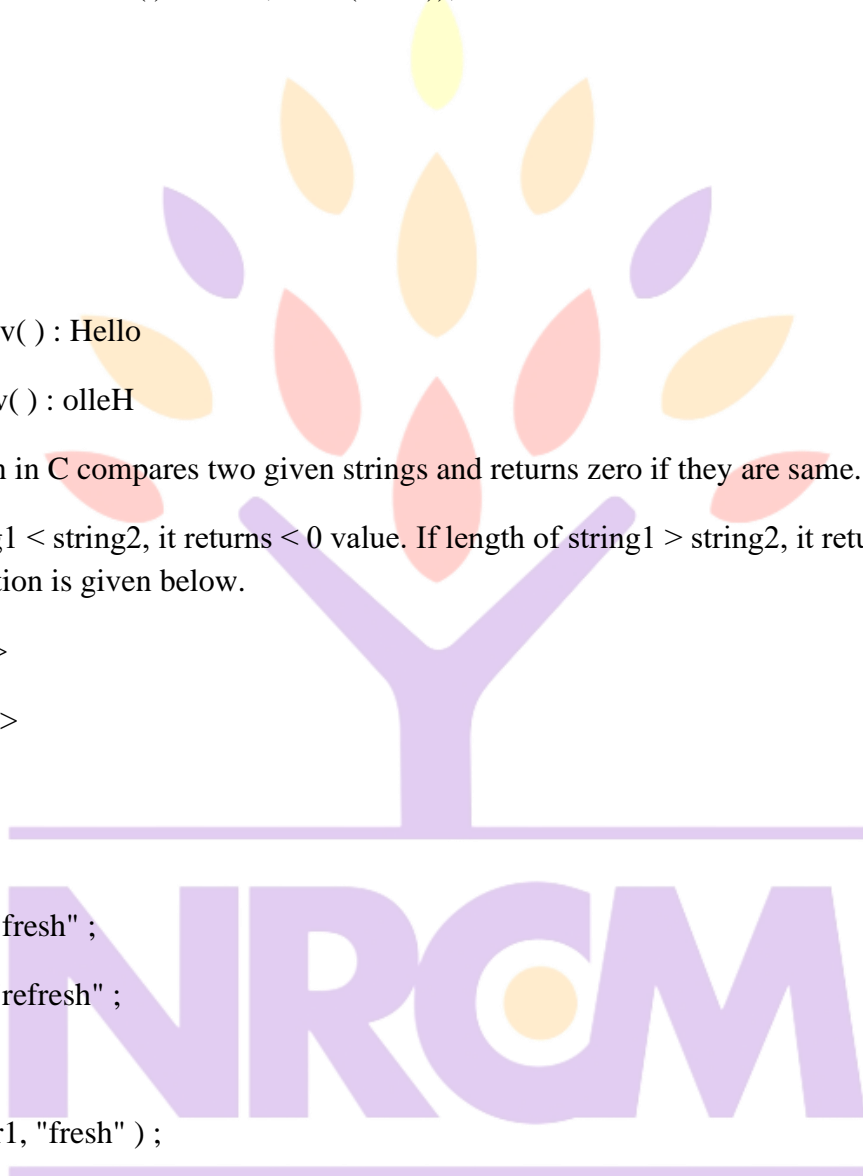
#include <stdio.h>
#include <string.h>
int main( )
{
char str1[ ] = "fresh" ;
char str2[ ] = "refresh" ;
int i, j, k ;
i = strcmp ( str1, "fresh" ) ;
j = strcmp ( str1, str2 ) ;
k = strcmp ( str1, "f" ) ;
printf ( "\n%d %d %d", i, j, k ) ;

return 0;
}

```

**Output:**

0 -1 1



your roots to success...

**strncmpi()** function in C is same as **strcmp()** function. But, **strncmpi()** function is not case sensitive. i.e, "A" and "a" are treated as same characters. Where as, **strcmp()** function treats "A" and "a" as different characters.

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

int main( )
{
    char str1[ ] = "fresh" ;
    char str2[ ] = "refresh" ;
    int i, j, k ;
    i = strncmpi ( str1, "FRESH" ) ;
    j = strcmp ( str1, str2 ) ;
    k = strcmp ( str1, "f" ) ;
    printf ( "\n%d %d %d", i, j, k ) ;
    return 0;
}
```

Output:

0 -1 1

**strchr():**

**strchr()** function returns pointer to the first occurrence of the character in a given string. Syntax for **strchr()** function is given below

```
#include <stdio.h>
#include <string.h>
int main ()
{
    char string[55] ="This is a string for testing";
    char *p;
    p = strchr (string,'i');
```

```

printf ("Character i is found at position %d\n",p-string+1);
printf ("First occurrence of character \'i\' in \'%s\' is \'%s\'",string, p);

return 0;
}

```

### Output:

Character i is found at position 3

First occurrence of character “i” in “This is a string for testing” is “is is a string for testing”

```

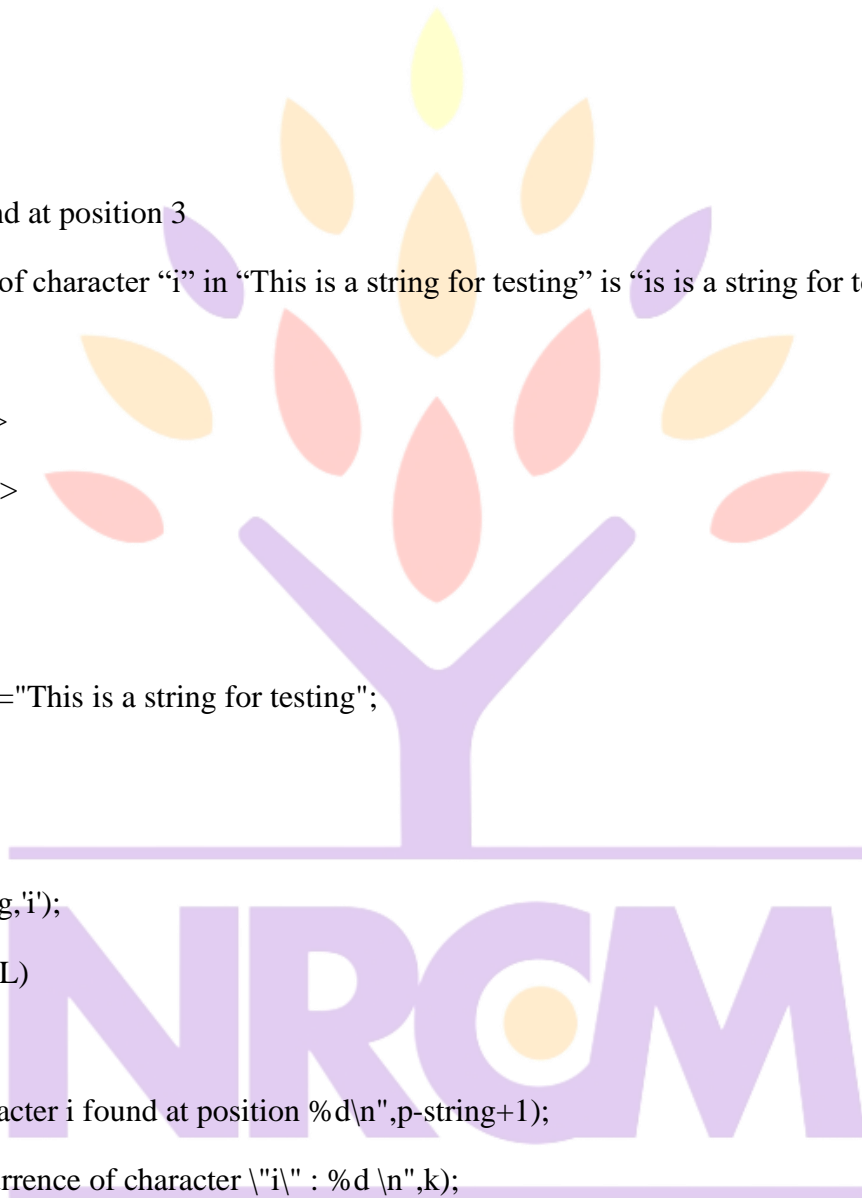
#include <stdio.h>
#include <string.h>
int main ()
{
char string[55] = "This is a string for testing";
char *p;
int k = 1;

p = strchr (string,'i');
while (p!=NULL)
{
printf ("Character i found at position %d\n",p-string+1);
printf ("Occurrence of character \'i\' : %d\n",k);
printf ("Occurrence of character \'i\' in \'%s\' is \'%s\' \
\\n",string, p);
p=strchr(p+1,'i');

k++;
}

return 0;

```



your roots to success...

```

}
Output:
Character i is found at position 3
Occurrence of character "i" : 1
Occurrence of character "i" in "This is a string for testing" is "is is a string for testing"
Character i is found at position 6
Occurrence of character "i" : 2
Occurrence of character "i" in "This is a string for testing" is "is a string for testing"
Character i is found at position 14
Occurrence of character "i" : 3
Occurrence of character "i" in "This is a string for testing" is "ing for testing"
Character i is found at position 26
Occurrence of character "i" : 4
Occurrence of character "i" in "This is a string for testing" is "ing"

```

**strchr());**

strchr ( ) last occurrence of given character in a string is found

```

#include <stdio.h>
#include <string.h>
int main ()
{
    char string[55] = "Hello world";
    char *p;
    p = strchr (string, 'l');

    printf ("Character i is found at position %d\n", p-string+1);
    printf ("last occurrence of character \"l\" in \"%s\" is \"%s\"", string, p);
}

```

```
return 0;  
}
```

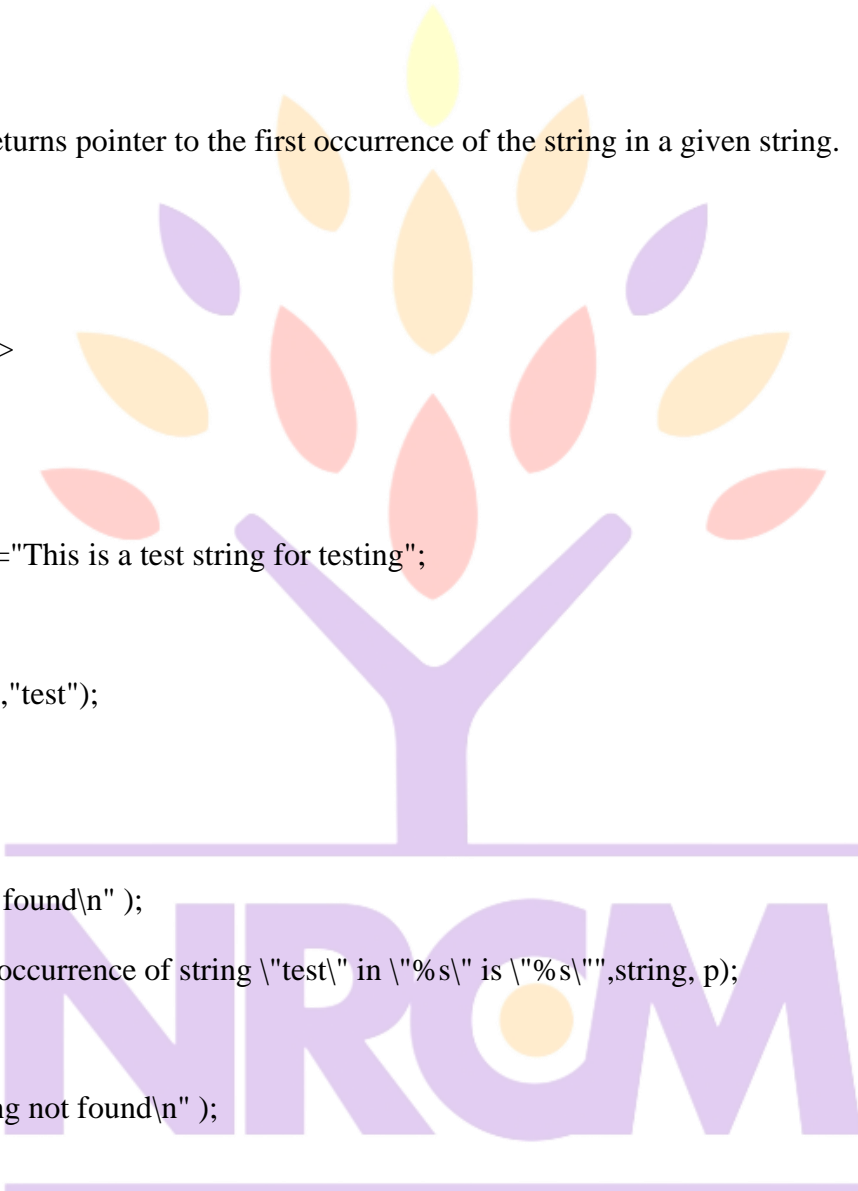
Character i is found at position 10

First occurrence of character ‘l’ in ‘Hello world’ is ‘ld’

### strstr():

strstr() function returns pointer to the first occurrence of the string in a given string.

```
include <stdio.h>  
#include <string.h>  
int main ()  
{  
char string[55] ="This is a test string for testing";  
char *p;  
p = strstr (string, "test");  
if(p)  
{  
printf("string found\n" );  
printf ("First occurrence of string \"test\" in \"%s\" is \"%s\"",string, p);  
}  
else printf("string not found\n" );  
return 0;  
}
```



your roots to success...

Output:

string found

First occurrence of string ‘test’ in ‘This is a test string for testing’ is ‘test string for testing’



## C – strdup() function

•strdup( ) function in C duplicates the given string.

```
#include <stdio.h>
#include <string.h>
int main()
{
    char *p1 = "Raja";
    char *p2;
    p2 = strdup(p1);
    printf("Duplicated string is : %s", p2);
    return 0;
}
```

Output:

Duplicated string is : Raja

**note** :strdup allocates memory for the new string on the heap, while using strcpy (or its safer strncpy variant) I can copy a string to a pre allocated memory on either the heap or the stack. char \*strdup(char \*pszSrc) ;  
strdup will allocate storage the size of the original string

## C – strset() function

•strset( ) function sets all the characters in a string to given character.

```
#include<stdio.h>
#include<string.h>
int main()
{
    char str[20] = "Test String";
    printf("Original string is : %s", str);
    printf("Test string after strset() : %s",strset(str,'#'));
}
```

```
printf("After string set: %s",str);  
return 0;  
}
```

Output:

```
Original string is          : Test String  
Test string after strset() : #####
```

### C – strnset() function

strnset( ) function sets portion of characters in a string to given.

strnset( ) function is non standard function which may not available in standard library in C.

```
#include<stdio.h>  
#include<string.h>  
int main()  
{  
    char str[20] = "Test String";  
    printf("Original string is : %s", str);  
    printf("Test string after string n set : %s", strnset(str,'#',4));  
    printf("After string n set : %s", str);  
    return 0;  
}
```

Output:

```
Original string is          : Test String  
Test string after string set : #### String
```

### C – strtok() function

strtok( ) function in C tokenizes/parses the given string using delimiter.

```
#include <stdio.h>
#include <string.h>
int main ()
{
    char string[50] = "Test,string1,Test,string2:Test:string3";
    char *p;
    printf ("String  \\"%s\\" is split into tokens:\n",string);
    p = strtok (string, ",:");
    while (p!= NULL)
    {
        printf ("%s\n",p);
        p = strtok (NULL, ",:");
    }
    return 0;
}
```

Output:

String "Test,string1,Test,string2:Test:string3" is split into tokens:

Test

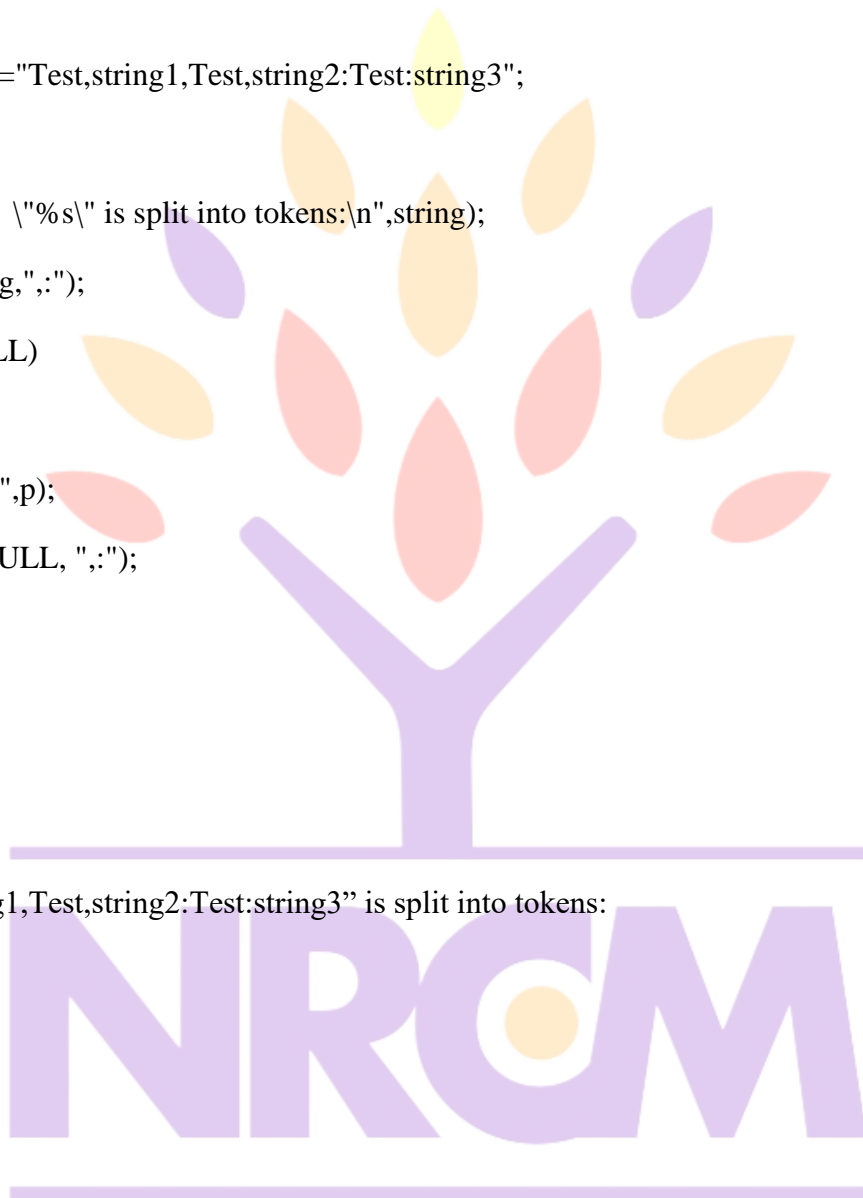
string1

Test

string2

Test

string3



your roots to success...

### Introduction to Data Structures

Data Structure is a way of collecting and organising data in such a way that we can perform operations on these data in an effective way. Data Structures is about rendering data elements in terms of some relationship, for better organization and storage. For example, we have data player's name "Virat" and age 26. Here "Virat" is of **String** data type and 26 is of **integer** data type.

We can organize this data as a record like **Player** record. Now we can collect and store player's records in a file or database as a data structure. For example: "Dhoni" 30, "Gambhir" 31, "Sehwag" 33

In simple language, Data Structures are structures programmed to store ordered data, so that various operations can be performed on it easily.

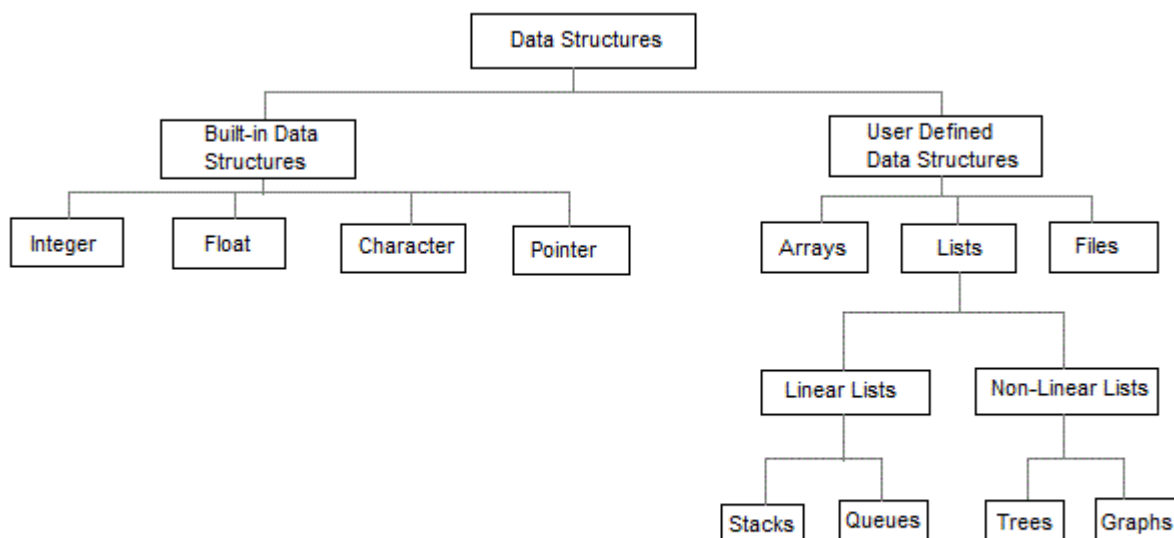
### Basic types of Data Structures

As we discussed above, anything that can store data can be called as a data structure, hence Integer, Float, Boolean, Char etc, all are data structures. They are known as **Primitive Data Structures**.

Then we also have some complex Data Structures, which are used to store large and connected data. Some example of **Abstract Data Structure** are :

- Linked List
- Tree
- Graph
- Stack, Queue etc.

All these data structures allow us to perform different operations on data. We select these data structures based on which type of operation is required. We will look into these data structures in more details in our later lessons.



## INTRODUCTION TO DATA STRUCTURES

### What is Algorithm ?

An algorithm is a finite set of instructions or logic, written in order, to accomplish a certain predefined task. Algorithm is not the complete code or program, it is just the core logic(solution) of a problem, which can be expressed either as an informal high level description as **pseudocode** or using a **flowchart**.

An algorithm is said to be efficient and fast, if it takes less time to execute and consumes less memory space. The performance of an algorithm is measured on the basis of following properties :

1. Time Complexity

## 2. Space Complexity

---

### Space Complexity

It's the amount of memory space required by the algorithm, during the course of its execution. Space complexity must be taken seriously for multi-user systems and in situations where limited memory is available.

An algorithm generally requires space for following components :

- **Instruction Space** : It's the space required to store the executable version of the program. This space is fixed, but varies depending upon the number of lines of code in the program.
  - **Data Space** : It's the space required to store all the constants and variables value.
  - **Environment Space** : It's the space required to store the environment information needed to resume the suspended function.
- 

### Time Complexity

Time Complexity is a way to represent the amount of time needed by the program to run to completion. We will study this in detail.

#### Time Complexity of Algorithms

Time complexity of an algorithm signifies the total time required by the program to run to completion. The time complexity of algorithms is most commonly expressed using the **big O notation**.

Time Complexity is most commonly estimated by counting the number of elementary functions performed by the algorithm. And since the algorithm's performance may vary with different types of input data, hence for an algorithm we usually use the **worst-case Time complexity** of an algorithm because that is the maximum time taken for any input size.

---

### Calculating Time Complexity

Now let's tap onto the next big topic related to Time complexity, which is How to Calculate Time Complexity. It becomes very confusing some times, but we will try to explain it in the simplest way.

Now the most common metric for calculating time complexity is Big O notation. This removes all constant factors so that the running time can be estimated in relation to **N**, as **N** approaches infinity. In general you can think of it like this :

statement;

Above we have a single statement. Its Time Complexity will be **Constant**. The running time of the statement will not change in relation to **N**.

```
for(i=0; i < N; i++)
```

```
{  
    statement;  
}
```

The time complexity for the above algorithm will be **Linear**. The running time of the loop is directly proportional to N. When N doubles, so does the running time.

```
for(i=0; i < N; i++)  
{  
    for(j=0; j < N;j++)  
    {  
        statement;  
    }  
}
```

This time, the time complexity for the above code will be **Quadratic**. The running time of the two loops is proportional to the square of N. When N doubles, the running time increases by  $N * N$ .

```
while(low <= high)  
{  
    mid = (low + high) / 2;  
    if (target < list[mid])  
        high = mid - 1;  
    else if (target > list[mid])  
        low = mid + 1;  
    else break;  
}
```

This is an algorithm to break a set of numbers into halves, to search a particular field(we will study this in detail later). Now, this algorithm will have a **Logarithmic** Time Complexity. The running time of the algorithm is proportional to the number of times N can be divided by 2(N is high-low here). This is because the algorithm divides the working area in half with each iteration.

your roots to success...

```
void quicksort(int list[], int left, int right)  
{  
    int pivot = partition(list, left, right);  
    quicksort(list, left, pivot - 1);  
    quicksort(list, pivot + 1, right);  
}
```

}

Taking the previous algorithm forward, above we have a small logic of Quick Sort (we will study this in detail later). Now in Quick Sort, we divide the list into halves every time, but we repeat the iteration  $N$  times (where  $N$  is the size of list). Hence time complexity will be  $N \cdot \log(N)$ . The running time consists of  $N$  loops (iterative or recursive) that are logarithmic, thus the algorithm is a combination of linear and logarithmic.

**NOTE :** In general, doing something with every item in one dimension is linear, doing something with every item in two dimensions is quadratic, and dividing the working area in half is logarithmic.

## Types of Notations for Time Complexity

Now we will discuss and understand the various notations used for Time Complexity.

1. **Big Oh** denotes "*fewer than or the same as*" <expression> iterations.
2. **Big Omega** denotes "*more than or the same as*" <expression> iterations.
3. **Big Theta** denotes "*the same as*" <expression> iterations.
4. **Little Oh** denotes "*fewer than*" <expression> iterations.
5. **Little Omega** denotes "*more than*" <expression> iterations.

## Understanding Notations of Time Complexity with Example

**O(expression)** is the set of functions that grow slower than or at the same rate as expression.

**Omega(expression)** is the set of functions that grow faster than or at the same rate as expression.

**Theta(expression)** consist of all the functions that lie in both  $O(\text{expression})$  and  $\Omega(\text{expression})$ .

Suppose you've calculated that an algorithm takes  $f(n)$  operations, where,

$$f(n) = 3n^2 + 2n + 4. \quad // \quad n^2 \text{ means square of } n$$

Since this polynomial grows at the same rate as  $n^2$ , then you could say that the function  $f$  lies in the set  $\Theta(n^2)$ . (It also lies in the sets  $O(n^2)$  and  $\Omega(n^2)$  for the same reason.)

The simplest explanation is, because **Theta** denotes *the same as* the expression. Hence, as  $f(n)$  grows by a factor of  $n^2$ , the time complexity can be best represented as  $\Theta(n^2)$ .

## Introduction to Sorting

Sorting is nothing but storage of data in sorted order, it can be in ascending or descending order. The term Sorting comes into picture with the term Searching. There are so many things in our real life that we need to search, like a particular record in database, roll numbers in merit list, a particular telephone number, any particular page in a book etc.

**Sorting** arranges data in a sequence which makes searching easier. Every record which is going to be sorted will contain one key. Based on the key the record will be sorted. For example, suppose we have a record of students, every such record will have the following data:

- Roll No.
- Name
- Age
- Class

Here Student roll no. can be taken as key for sorting the records in ascending or descending order. Now suppose we have to search a Student with roll no. 15, we don't need to search the complete record we will simply search between the Students with roll no. 10 to 20.

### Sorting Efficiency

There are many techniques for sorting. Implementation of particular sorting technique depends upon situation. Sorting techniques mainly depends on two parameters. First parameter is the execution time of program, which means time taken for execution of program. Second is the space, which means space taken by the program.

### Types of Sorting Techniques

There are many types of Sorting techniques, differentiated by their efficiency and space requirements. Following are some sorting techniques which we will be covering in next sections.

1. Bubble Sort
2. Insertion Sort
3. Selection Sort
4. Quick Sort
5. Merge Sort
6. Heap Sort

### Bubble Sorting

**Bubble Sort** is an algorithm which is used to sort **N** elements that are given in a memory for eg: an Array with **N** number of elements. Bubble Sort compares all the element one by one and sort them based on their values.

It is called Bubble sort, because with each iteration the smaller element in the list bubbles up towards the first place, just like a water bubble rises up to the water surface.

Sorting takes place by stepping through all the data items one-by-one in pairs and comparing adjacent data items and swapping each pair that is out of order.

5	1	6	2	4	3
---	---	---	---	---	---

Lets take this Array.

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

Here we can see the Array after the first iteration.

Similarly, after other consecutive iterations, this array will get sorted.

### Sorting using Bubble Sort Algorithm

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



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

```
int i, j, temp;
```

```
for(i=0; i<6, i++)
```

```
{  
    for(j=0; j<6-i-1; j++)  
    {  
        if( a[j] > a[j+1])  
        {  
            temp = a[j];  
            a[j] = a[j+1];  
            a[j+1] = temp;  
        }  
    }  
}
```

```
//now you can print the sorted array after this
```

Above is the algorithm, to sort an array using Bubble Sort. Although the above logic will sort and unsorted array, still the above algorithm isn't efficient and can be enhanced further. Because as per the above logic, the for loop will keep going for six iterations even if the array gets sorted after the second iteration.

Hence we can insert a flag and can keep checking whether swapping of elements is taking place or not. If no swapping is taking place that means the array is sorted and we can jump out of the for loop.

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

```
int i, j, temp;
```

```
for(i=0; i<6, i++)
```

```
{  
    for(j=0; j<6-i-1; j++)  
    {  
        int flag = 0;           //taking a flag variable  
        if( a[j] > a[j+1])  
        {  
            temp = a[j];  
            a[j] = a[j+1];  
            a[j+1] = temp;  
            flag = 1;           //setting flag as 1, if swapping occurs  
        }  
    }  
    if(!flag)                   //breaking out of for loop if no swapping takes place  
    {  
        break;  
    }  
}
```

```

}
}

```

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

### Complexity Analysis of Bubble Sorting

In Bubble Sort, n-1 comparisons will be done in 1st pass, n-2 in 2nd pass, n-3 in 3rd pass and so on. So the total number of comparisons will be

$$(n-1)+(n-2)+(n-3)+.....+3+2+1$$

$$\text{Sum} = n(n-1)/2$$

i.e  $O(n^2)$

Hence the complexity of Bubble Sort is  $O(n^2)$ .

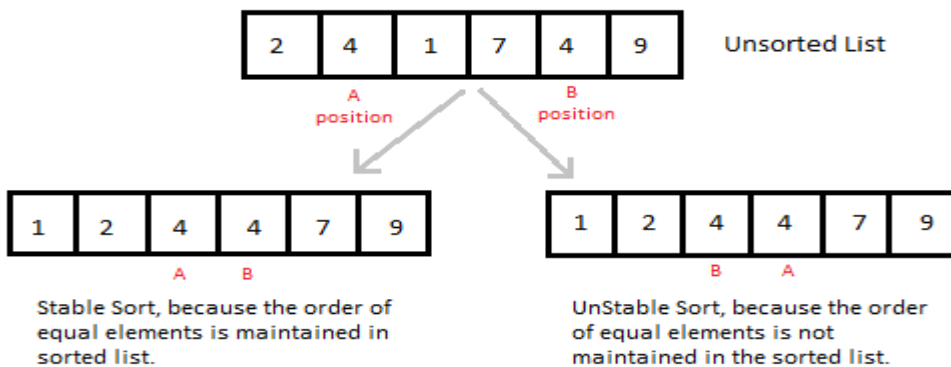
The main advantage of Bubble Sort is the simplicity of the algorithm.Space complexity for Bubble Sort is  $O(1)$ , because only single additional memory space is required for **temp** variable

**Best-case** Time Complexity will be  $O(n)$ , it is when the list is already sorted.

### Insertion Sorting

It is a simple Sorting algorithm which sorts the array by shifting elements one by one. Following are some of the important characteristics of Insertion Sort.

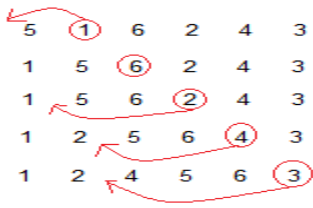
1. It has one of the simplest implementation
2. It is efficient for smaller data sets, but very inefficient for larger lists.
3. Insertion Sort is adaptive, that means it reduces its total number of steps if given a partially sorted list, hence it increases its efficiency.
4. It is better than Selection Sort and Bubble Sort algorithms.
5. Its space complexity is less, like Bubble Sorting, insertion sort also requires a single additional memory space.
6. It is **Stable**, as it does not change the relative order of elements with equal keys



### How Insertion Sorting Works

5	1	6	2	4	3
---	---	---	---	---	---

Lets take this Array.



( Always we start with the second element as key.)

As we can see here, in insertion sort, we pick up a key, and compares it with elements ahead of it, and puts the key in the right place

5 has nothing before it.

1 is compared to 5 and is inserted before 5.

6 is greater than 5 and 1.

2 is smaller than 6 and 5, but greater than 1, so its is inserted after 1.

And this goes on...

## Sorting using Insertion Sort Algorithm

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

```
int i, j, key;
```

```
for(i=1; i<6; i++)
```

```
{
```

```
    key = a[i];
```

```
    j = i-1;
```

```
    while(j>=0 && key < a[j])
```

```
    {
```

```
        a[j+1] = a[j];
```

```
        j--;
```

```
    }
```

```
    a[j+1] = key;
```

```
}
```

Now lets, understand the above simple insertion sort algorithm. We took an array with 6 integers. We took a variable **key**, in which we put each element of the array, in each pass, starting from the second element, that is **a[1]**.

Then using the while loop, we iterate, until **j** becomes equal to zero or we find an element which is greater than **key**, and then we insert the key at that position.

In the above array, first we pick 1 as key, we compare it with 5(element before 1), 1 is smaller than 5, we shift 1 before 5. Then we pick 6, and compare it with 5 and 1, no shifting this time. Then 2 becomes the key and is compared with, 6 and 5, and then 2 is placed after 1. And this goes on, until complete array gets sorted.

## Complexity Analysis of Insertion Sorting

**Worst Case Time Complexity :**  $O(n^2)$

**Best Case Time Complexity :**  $O(n)$

**Average Time Complexity :**  $O(n^2)$

**Space Complexity :**  $O(1)$  Selection Sorting

Selection sorting is conceptually the most simplest sorting algorithm. This algorithm first finds the smallest element in the array and exchanges it with the element in the first position, then find the second smallest

element and exchange it with the element in the second position, and continues in this way until the entire array is sorted.

### How Selection Sorting Works

Original Array	After 1st pass	After 2nd pass	After 3rd pass	After 4th pass	After 5th pass
3	1	1	1	1	1
6	6	3	3	3	3
1	3	6	4	4	4
8	8	8	8	5	5
4	4	4	6	6	6
5	5	5	5	8	8

In the first pass, the smallest element found is 1, so it is placed at the first position, then leaving first element, smallest element is searched from the rest of the elements, 3 is the smallest, so it is then placed at the second position. Then we leave 1 and 3, from the rest of the elements, we search for the smallest and put it at third position and keep doing this, until array is sorted.

### Sorting using Selection Sort Algorithm

```
void selectionSort(int a[], int size)
{
    int i, j, min, temp;
    for(i=0; i < size-1; i++)
    {
        min = i; //setting min as i
        for(j=i+1; j < size; j++)
        {
            if(a[j] < a[min]) //if element at j is less than element at min position
            {
                min = j; //then set min as j
            }
        }
        temp = a[i];
        a[i] = a[min];
        a[min] = temp;
    }
}
```

### Complexity Analysis of Selection Sorting

**Worst Case Time Complexity :**  $O(n^2)$

**Best Case Time Complexity :**  $O(n^2)$

**Average Time Complexity :**  $O(n^2)$

**Space Complexity :**  $O(1)$

### Quick Sort Algorithm

Quick Sort, as the name suggests, sorts any list very quickly. Quick sort is not stable search, but it is very fast and requires very less additional space. It is based on the rule of **Divide and Conquer**(also called *partition-exchange sort*). This algorithm divides the list into three main parts :

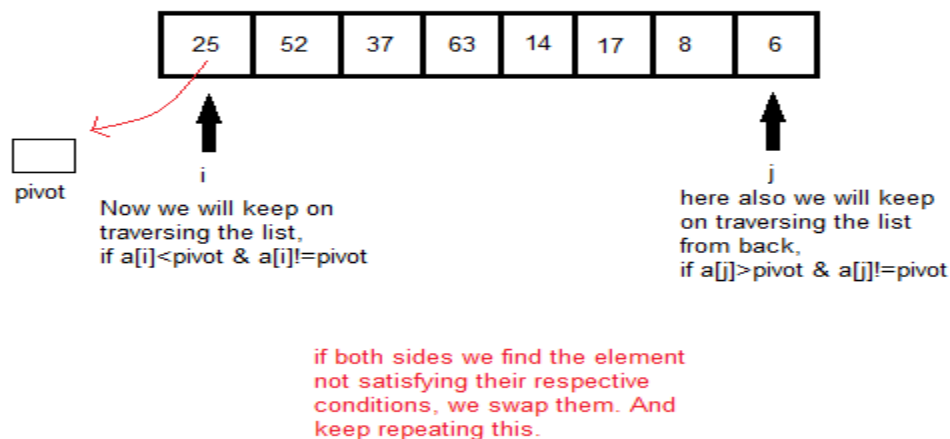
1. Elements less than the Pivot element
2. Pivot element
3. Elements greater than the pivot element

In the list of elements, mentioned in below example, we have taken **25** as pivot. So after the first pass, the list will be changed like this.

6 8 17 14 25 63 37 52

Hence after the first pass, pivot will be set at its position, with all the elements smaller to it on its left and all the elements larger than it on the right. Now 6 8 17 14 and 63 37 52 are considered as two separate lists, and same logic is applied on them, and we keep doing this until the complete list is sorted.

### How Quick Sorting Works



### DIVIDE AND CONQUER - QUICK SORT

### Sorting using Quick Sort Algorithm

```
/* a[] is the array, p is starting index, that is 0,  
and r is the last index of array. */
```

```
void quicksort(int a[], int p, int r)  
{  
    if(p < r)  
    {  
        int q;
```

```
    q = partition(a, p, r);
    quicksort(a, p, q);
    quicksort(a, q+1, r);
}
}
```

```
int partition(int a[], int p, int r)
{
    int i, j, pivot, temp;
    pivot = a[p];
    i = p;
    j = r;
    while(1)
    {
        while(a[i] < pivot && a[i] != pivot)
            i++;
        while(a[j] > pivot && a[j] != pivot)
            j--;
        if(i < j)
        {
            temp = a[i];
            a[i] = a[j];
            a[j] = temp;
        }
        else
        {
            return j;
        }
    }
}
```

---

your roots to success...

### **Complexity Analysis of Quick Sort**

**Worst Case Time Complexity :**  $O(n^2)$

**Best Case Time Complexity :**  $O(n \log n)$

**Average Time Complexity :**  $O(n \log n)$

**Space Complexity :**  $O(n \log n)$

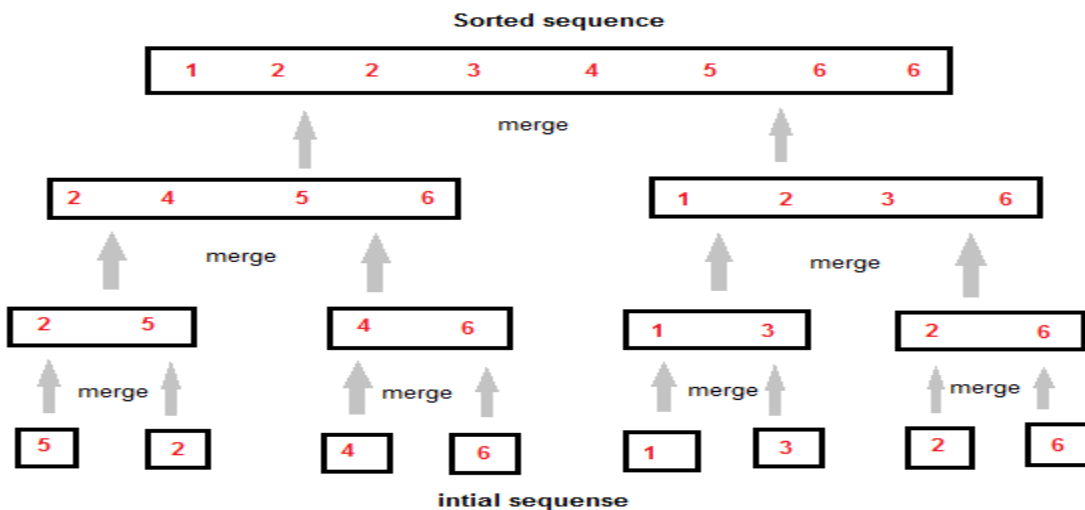
- Space required by quick sort is very less, only  $O(n \log n)$  additional space is required.
- Quick sort is not a stable sorting technique, so it might change the occurrence of two similar elements in the list while sorting.

### Merge Sort Algorithm

Merge Sort follows the rule of **Divide and Conquer**. But it doesn't divide the list into two halves. In merge sort the unsorted list is divided into  $N$  sublists, each having one element, because a list of one element is considered sorted. Then, it repeatedly merge these sublists, to produce new sorted sublists, and at last one sorted list is produced.

Merge Sort is quite fast, and has a time complexity of  $O(n \log n)$ . It is also a stable sort, which means the "equal" elements are ordered in the same order in the sorted list.

### How Merge Sort Works



Like we can see in the above example, merge sort first breaks the unsorted list into sorted sublists, and then keep merging these sublists, to finally get the complete sorted list.

### Sorting using Merge Sort Algorithm

```
/* a[] is the array, p is starting index, that is 0,
and r is the last index of array. */
```

Lets take  $a[5] = \{32, 45, 67, 2, 7\}$  as the array to be sorted.

```
void mergesort(int a[], int p, int r)
```

```
{  
    int q;  
    if(p < r)  
    {  
        q = floor( (p+r) / 2);  
        mergesort(a, p, q);  
        mergesort(a, q+1, r);  
        merge(a, p, q, r);  
    }  
}
```

```
void merge(int a[], int p, int q, int r)
```

```
{  
    int b[5];    //same size of a[]  
    int i, j, k;  
    k = 0;  
    i = p;  
    j = q+1;  
    while(i <= q && j <= r)  
    {  
        if(a[i] < a[j])  
        {  
            b[k++] = a[i++];    // same as b[k]=a[i]; k++; i++;  
        }  
        else  
        {  
            b[k++] = a[j++];  
        }  
    }  
  
    while(i <= q)  
    {  
        b[k++] = a[i++];  
    }  
  
    while(j <= r)
```



```

{
    b[k++] = a[j++];
}

for(i=r; i >= p; i--)
{
    a[i] = b[--k];        // copying back the sorted list to a[]
}
}

```

### Complexity Analysis of Merge Sort

**Worst Case Time Complexity :**  $O(n \log n)$

**Best Case Time Complexity :**  $O(n \log n)$

**Average Time Complexity :**  $O(n \log n)$

**Space Complexity :**  $O(n)$

- Time complexity of Merge Sort is  $O(n \log n)$  in all 3 cases (worst, average and best) as merge sort always divides the array in two halves and take linear time to merge two halves.
- It requires equal amount of additional space as the unsorted list. Hence its not at all recommended for searching large unsorted lists.
- It is the best Sorting technique for sorting **Linked Lists**.

### Heap Sort Algorithm

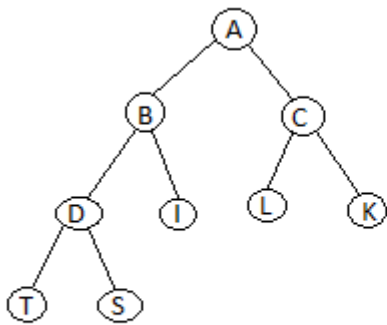
Heap Sort is one of the best sorting methods being in-place and with no quadratic worst-case scenarios. Heap sort algorithm is divided into two basic parts :

- Creating a Heap of the unsorted list.
- Then a sorted array is created by repeatedly removing the largest/smallest element from the heap, and inserting it into the array. The heap is reconstructed after each removal.

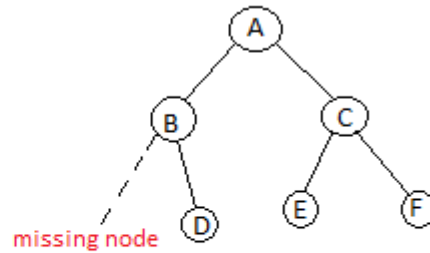
### What is a Heap ?

Heap is a special tree-based data structure, that satisfies the following special heap properties :

1. **Shape Property :** Heap data structure is always a Complete Binary Tree, which means all levels of the tree are fully filled.

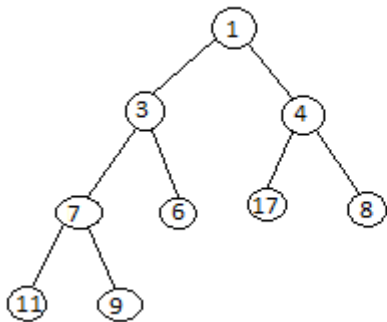


Complete Binary Tree



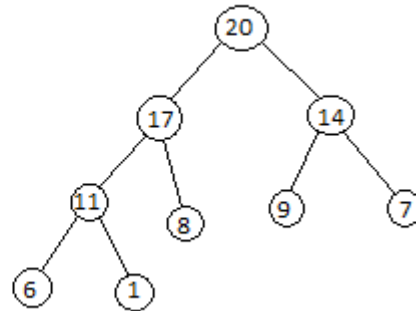
In-Complete Binary Tree

2. **Heap Property :** All nodes are either [greater than or equal to] or [less than or equal to] each of its children. If the parent nodes are greater than their children, heap is called a **Max-Heap**, and if the parent nodes are smaller than their child nodes, heap is called **Min-Heap**.



Min-Heap

In min-heap, first element is the smallest. So when we want to sort a list in ascending order, we create a Min-heap from that list, and pick the first element, as it is the smallest, then we repeat the process with remaining elements.



Max-Heap

In max-heap, the first element is the largest, hence it is used when we need to sort a list in descending order.

## How Heap Sort Works

Initially on receiving an unsorted list, the first step in heap sort is to create a Heap data structure (Max-Heap or Min-Heap). Once heap is built, the first element of the Heap is either largest or smallest (depending upon Max-Heap or Min-Heap), so we put the first element of the heap in our array. Then we again make heap using the remaining elements, to again pick the first element of the heap and put it into the array. We keep on doing the same repeatedly until we have the complete sorted list in our array.

In the below algorithm, initially **heapsort()** function is called, which calls **buildheap()** to build heap, which in turn uses **satisfyheap()** to build the heap.

## Sorting using Heap Sort Algorithm

```
/* Below program is written in C++ language */

void heapsort(int[], int);
void buildheap(int [], int);
void satisfyheap(int [], int, int);

void main()
{
    int a[10], i, size;
    cout <<"Enter size of list";    // less than 10, because max size of array is 10
    cin >> size;
    cout <<"Enter"<< size <<"elements";
    for( i=0; i < size; i++)
    {
        cin >> a[i];
    }
    heapsort(a, size);
    getch();
}

void heapsort(int a[], int length)
{
    buildheap(a, length);
    int heapsize, i, temp;
    heapsize = length - 1;
    for( i=heapsize; i >= 0; i--)
    {
        temp = a[0];
        a[0] = a[heapsize];
        a[heapsize] = temp;
        heapsize--;
        satisfyheap(a, 0, heapsize);
    }
    for( i=0; i < length; i++)
    {
        cout <<"\t"<< a[i];
    }
}
```

```

    }
}

void buildheap(int a[], int length)
{
    int i, heapsize;
    heapsize = length - 1;
    for( i=(length/2); i >= 0; i--)
    {
        satisfyheap(a, i, heapsize);
    }
}

void satisfyheap(int a[], int i, int heapsize)
{
    int l, r, largest, temp;
    l = 2*i;
    r = 2*i + 1;
    if(l <= heapsize && a[l] > a[i])
    {
        largest = l;
    }
    else
    {
        largest = i;
    }
    if( r <= heapsize && a[r] > a[largest])
    {
        largest = r;
    }
    if(largest != i)
    {
        temp = a[i];
        a[i] = a[largest];
        a[largest] = temp;
        satisfyheap(a, largest, heapsize);
    }
}

```

```
}
```

### Complexity Analysis of Heap Sort

**Worst Case Time Complexity :**  $O(n \log n)$

**Best Case Time Complexity :**  $O(n \log n)$

**Average Time Complexity :**  $O(n \log n)$

**Space Complexity :**  $O(n)$

- Heap sort is not a Stable sort, and requires a constant space for sorting a list.
- Heap Sort is very fast and is widely used for sorting.

### Searching Algorithms on Array

Before studying searching algorithms on array we should know what is an algorithm?

An **algorithm** is a step-by-step procedure or method for solving a problem by a computer in a given number of steps. The steps of an algorithm may include repetition depending upon the problem for which the algorithm is being developed. The algorithm is written in human readable and understandable form. To search an element in a given array, it can be done in two ways Linear search and Binary search.

#### Linear Search

A linear search is the basic and simple search algorithm. A linear search searches an element or value from an array till the desired element or value is not found and it searches in a sequence order. It compares the element with all the other elements given in the list and if the element is matched it returns the value index else it return -1. Linear Search is applied on the unsorted or unordered list when there are fewer elements in a list.

#### Example with Implementation

To search the element 5 it will go step by step in a sequence order.

8	2	6	3	5
---	---	---	---	---

```
function findIndex(values, target)
{
    for(var i = 0; i < values.length; ++i)
    {
```

```
    if (values[i] == target)
    {
        return i;
    }
}
return -1;
}
```

*//call the function findIndex with array and number to be searched*

```
findIndex([ 8 , 2 , 6 , 3 , 5 ] , 5) ;
```

## Binary Search

Binary Search is applied on the sorted array or list. In binary search, we first compare the value with the elements in the middle position of the array. If the value is matched, then we return the value. If the value is less than the middle element, then it must lie in the lower half of the array and if it's greater than the element then it must lie in the upper half of the array. We repeat this procedure on the lower (or upper) half of the array. Binary Search is useful when there are large numbers of elements in an array.

## Example with Implementation

To search an element 13 from the sorted array or list.

2	4	7	9	13	15
---	---	---	---	----	----

- As we can see the above array is sorted in ascending order.
- Binary Search is applied on sorted lists only, so that we can make the search fast, by breaking the list everytime.
- Start with middle element,
- if its EQUAL to the number we are searching, then RETURN
- if its less than it, then move to the RIGHT.
- if its more that it, then move to the LEFT.
- And then, REPEAT, till you find the number.

```
function findIndex(values, target)
{
    return binarySearch(values, target, 0, values.length - 1);
};
```

```
function binarySearch(values, target, start, end) {
  if (start > end) { return -1; } //does not exist

  var middle = Math.floor((start + end) / 2);
  var value = values[middle];

  if (value > target) { return binarySearch(values, target, start, middle-1); }
  if (value < target) { return binarySearch(values, target, middle+1, end); }
  return middle; //found!
}
```

```
findIndex([2, 4, 7, 9, 13, 15], 13);
```

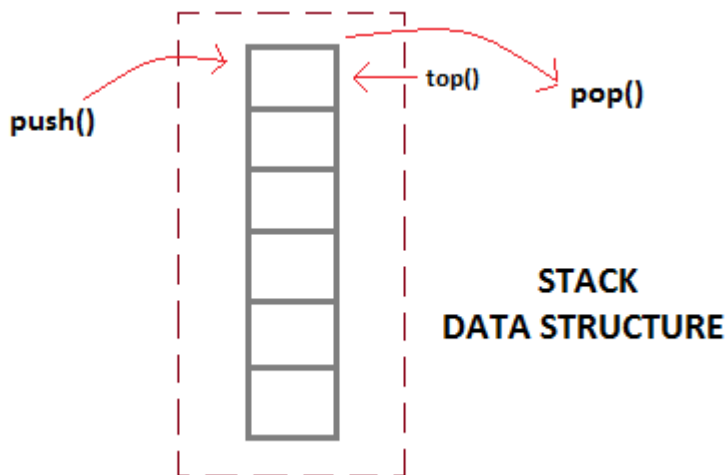
In the above program logic, we are first comparing the middle number of the list, with the target, if it matches we return. If it doesn't, we see whether the middle number is greater than or smaller than the target.

If the Middle number is greater than the Target, we start the binary search again, but this time on the left half of the list, that is from the start of the list to the middle, not beyond that.

If the Middle number is smaller than the Target, we start the binary search again, but on the right half of the list, that is from the middle of the list to the end of the list.

## Stacks

Stack is an abstract data type with a bounded(predefined) capacity. It is a simple data structure that allows adding and removing elements in a particular order. Every time an element is added, it goes on the top of the stack, the only element that can be removed is the element that was at the top of the stack, just like a pile of objects.



### Basic features of Stack

1. Stack is an ordered list of similar data type.
2. Stack is a **LIFO** structure. (Last in First out).
3. **push()** function is used to insert new elements into the Stack and **pop()** is used to delete an element from the stack. Both insertion and deletion are allowed at only one end of Stack called **Top**.
4. Stack is said to be in **Overflow** state when it is completely full and is said to be in **Underflow** state if it is completely empty.

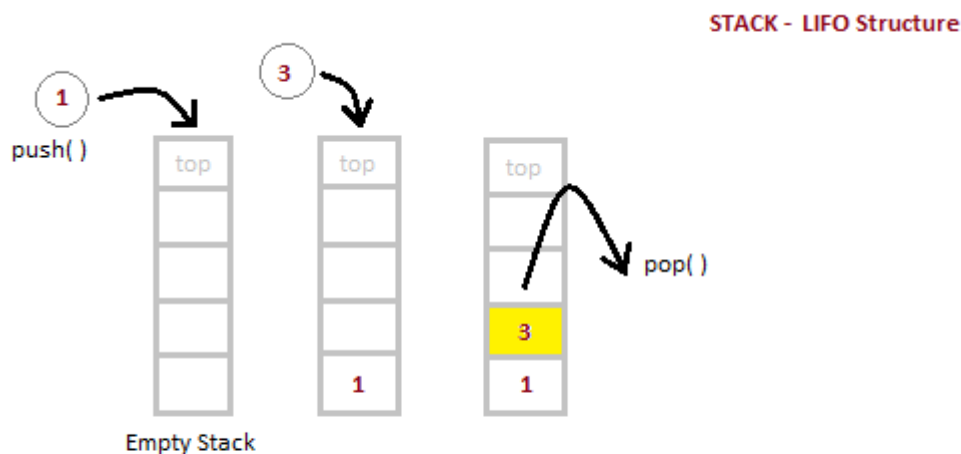
### Applications of Stack

The simplest application of a stack is to reverse a word. You push a given word to stack - letter by letter - and then pop letters from the stack.

There are other uses also like : **Parsing, Expression Conversion**(Infix to Postfix, Postfix to Prefix etc) and many more.

### Implementation of Stack

Stack can be easily implemented using an Array or a Linked List. Arrays are quick, but are limited in size and Linked List requires overhead to allocate, link, unlink, and deallocate, but is not limited in size. Here we will implement Stack using array.



In a Stack, all operations take place at the "top" of the stack. The "push" operation adds an item to the top of the Stack.  
The "pop" operation removes the item on top of the stack.

/\* Below program is written in C++ language \*/

```

Class Stack
{
    int top;
    public:
    int a[10];    //Maximum size of Stack
    Stack()
  
```



```
{
    top = -1;
}
};

void Stack::push(int x)
{
    if( top >= 10)
    {
        cout <<"Stack Overflow";
    }
    else
    {
        a[++top] = x;
        cout <<"Element Inserted";
    }
}

int Stack::pop()
{
    if(top < 0)
    {
        cout <<"Stack Underflow";
        return 0;
    }
    else
    {
        int d = a[--top];
        return d;
    }
}

void Stack::isEmpty()
{
    if(top < 0)
    {
        cout <<"Stack is empty";
    }
}
```

```

}
else
{
    cout <<"Stack is not empty";
}
}

```

Position of Top	Status of Stack
-1	Stack is Empty
0	Only one element in Stack
N-1	Stack is Full
N	Overflow state of Stack

### Analysis of Stacks

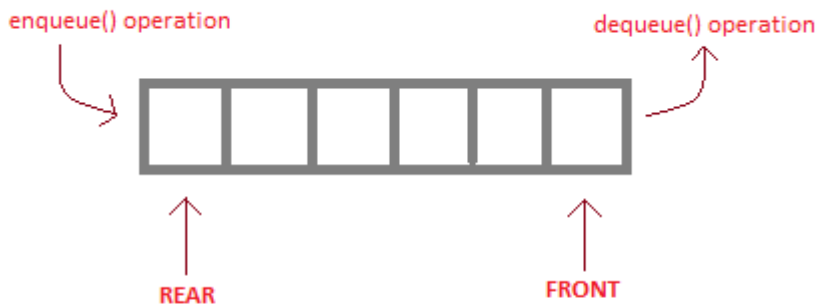
Below mentioned are the time complexities for various operations that can be performed on the Stack data structure.

- **Push Operation** :  $O(1)$
- **Pop Operation** :  $O(1)$
- **Top Operation** :  $O(1)$
- **Search Operation** :  $O(n)$

### Queue Data Structures

Queue is also an abstract data type or a linear data structure, in which the first element is inserted from one end called **REAR**(also called tail), and the deletion of existing element takes place from the other end called as **FRONT**(also called head). This makes queue as FIFO data structure, which means that element inserted first will also be removed first.

The process to add an element into queue is called **Enqueue** and the process of removal of an element from queue is called **Dequeue**.



**enqueue( )** is the operation for adding an element into Queue.

**dequeue( )** is the operation for removing an element from Queue .

### QUEUE DATA STRUCTURE

#### Basic features of Queue

1. Like Stack, Queue is also an ordered list of elements of similar data types.
2. Queue is a FIFO( First in First Out ) structure.
3. Once a new element is inserted into the Queue, all the elements inserted before the new element in the queue must be removed, to remove the new element.
4. **peek( )** function is oftenly used to return the value of first element without dequeuing it.

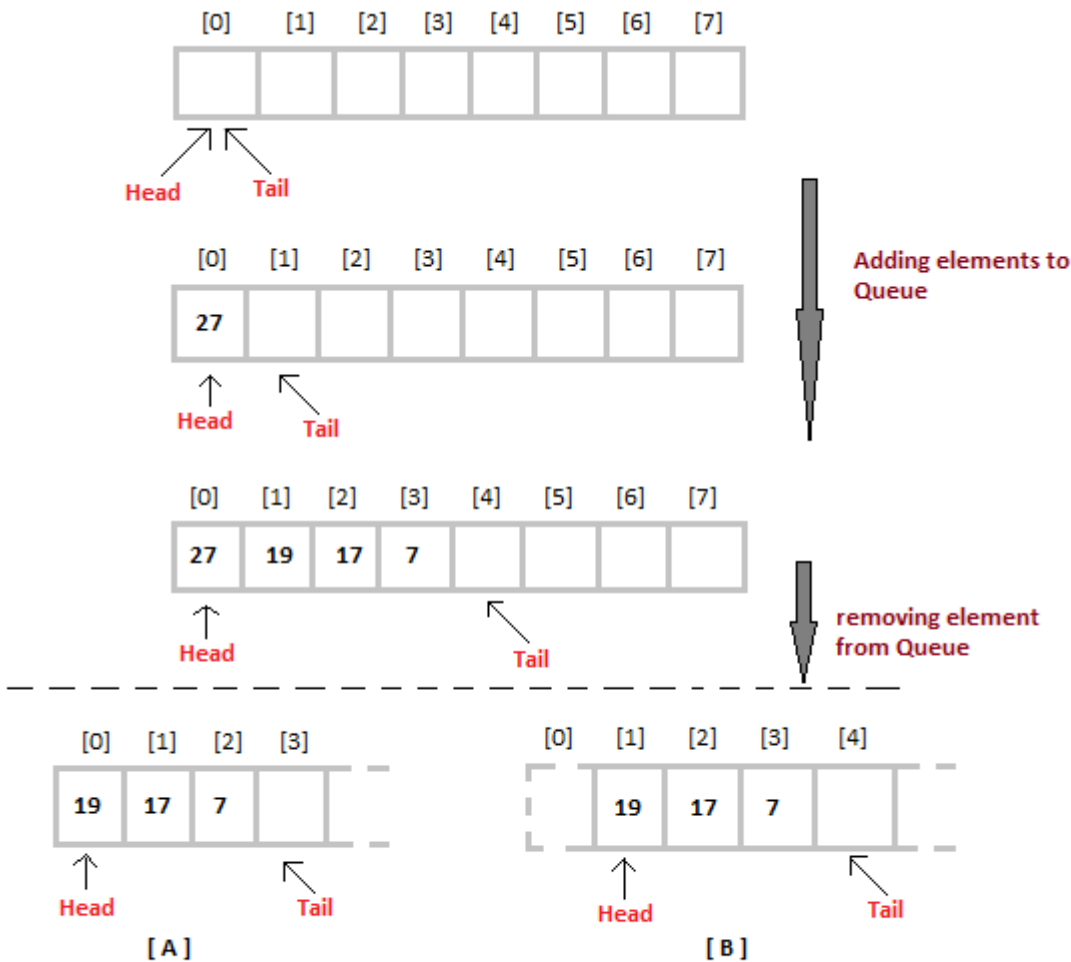
#### Applications of Queue

Queue, as the name suggests is used whenever we need to have any group of objects in an order in which the first one coming in, also gets out first while the others wait for there turn, like in the following scenarios :

1. Serving requests on a single shared resource, like a printer, CPU task scheduling etc.
2. In real life, Call Center phone systems will use Queues, to hold people calling them in an order, until a service representative is free.
3. Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive, First come first served.

#### Implementation of Queue

Queue can be implemented using an Array, Stack or Linked List. The easiest way of implementing a queue is by using an Array. Initially the **head**(FRONT) and the **tail**(REAR) of the queue points at the first index of the array (starting the index of array from 0). As we add elements to the queue, the tail keeps on moving ahead, always pointing to the position where the next element will be inserted, while the head remains at the first index.



When we remove element from Queue, we can follow two possible approaches (mentioned [A] and [B] in above diagram). In [A] approach, we remove the element at **head** position, and then one by one move all the other elements on position forward. In approach [B] we remove the element from **head** position and then move **head** to the next position.

In approach [A] there is an overhead of shifting the elements one position forward every time we remove the first element. In approach [B] there is no such overhead, but whenever we move head one position ahead, after removal of first element, the size of Queue is reduced by one space each time.

/\* Below program is written in C++ language \*/

```
#define SIZE 100
class Queue
{
    int a[100];
    int rear;    //same as tail
    int front;  //same as head

public:
```

```

Queue()
{
    rear = front = -1;
}
void enqueue(int x);    //declaring enqueue, dequeue and display functions
int dequeue();
void display();
}

void Queue :: enqueue(int x)
{
    if( rear = SIZE-1)
    {
        cout <<"Queue is full";
    }
    else
    {
        a[++rear] = x;
    }
}

int queue :: dequeue()
{
    return a[++front];    //following approach [B], explained above
}

void queue :: display()
{
    int i;
    for( i = front; i <= rear; i++)
    {
        cout << a[i];
    }
}

```

To implement approach [A], you simply need to change the dequeue method, and include a for loop which will shift all the remaining elements one position.

```

return a[0];    //returning first element
for (i = 0; i < tail-1; i++)    //shifting all other elements
{
    a[i]= a[i+1];
}

```

```
tail--;  
}
```

### Analysis of Queue

- Enqueue : **O(1)**
- Dequeue : **O(1)**
- Size : **O(1)**

### Queue Data Structure using Stack

A Queue is defined by its property of FIFO, which means First in First Out, i.e the element which is added first is taken out first. Hence we can implement a Queue using Stack for storage instead of array.

For performing **enqueue** we require only one stack as we can directly **push** data into stack, but to perform **dequeue** we will require two Stacks, because we need to follow queue's FIFO property and if we directly **pop** any data element out of Stack, it will follow LIFO approach (Last in First Out).

### Implementation of Queue using Stacks

In all we will require two Stacks, we will call them InStack and OutStack.

```
class Queue {  
    public:  
    Stack S1, S2;  
    //defining methods  
    void enqueue(int x);  
  
    int dequeue();  
}
```

We know that, Stack is a data structure, in which data can be added using **push()** method and data can be deleted using **pop()** method. To learn about Stack, follow the link : [Stack Data Structure](#)

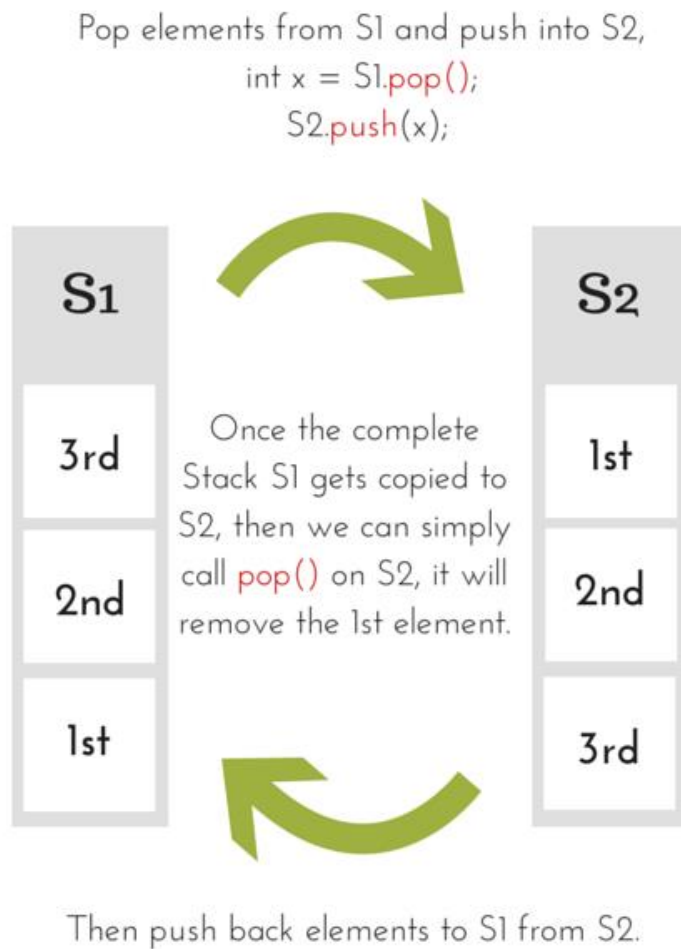
### Adding Data to Queue

As our Queue has Stack for data storage in place of arrays, hence we will be adding data to Stack, which can be done using the **push()** method, hence :

```
void Queue :: enqueue(int x) {  
    S1.push(x);  
}
```

### Removing Data from Queue

When we say remove data from Queue, it always means taking out the First element first and so on, as we have to follow the FIFO approach. But if we simply perform `S1.pop()` in our **dequeue** method, then it will remove the Last element first. So what to do now?



```
int Queue :: dequeue() {  
    while(S1.isEmpty()) {  
        x = S1.pop();  
        S2.push();  
    }  
  
    //removing the element  
    x = S2.pop();  
  
    while(!S2.isEmpty()) {  
        x = S2.pop();  
        S1.push(x);  
    }  
}
```

```
return x;  
}
```

## Introduction to Linked Lists

Linked List is a linear data structure and it is very common data structure which consists of group of nodes in a sequence which is divided in two parts. Each node consists of its own data and the address of the next node and forms a chain. Linked Lists are used to create trees and graphs.



## Advantages of Linked Lists

- They are a dynamic in nature which allocates the memory when required.
- Insertion and deletion operations can be easily implemented.
- Stacks and queues can be easily executed.
- Linked List reduces the access time.

## Disadvantages of Linked Lists

- The memory is wasted as pointers require extra memory for storage.
- No element can be accessed randomly; it has to access each node sequentially.
- Reverse Traversing is difficult in linked list.

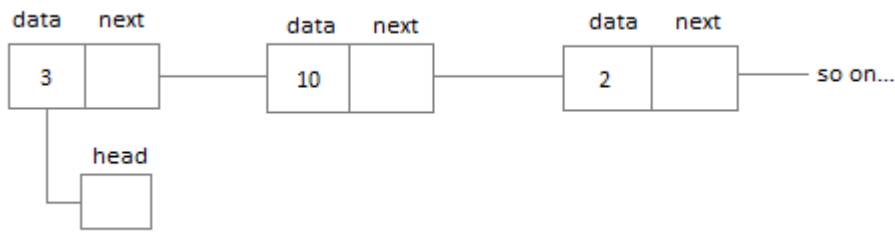
## Applications of Linked Lists

- Linked lists are used to implement stacks, queues, graphs, etc.
- Linked lists let you insert elements at the beginning and end of the list.
- In Linked Lists we don't need to know the size in advance.

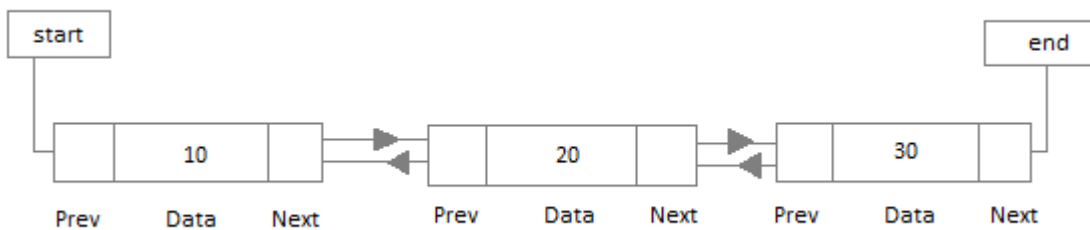
## Types of Linked Lists

- **Singly Linked List** : Singly linked lists contain nodes which have a data part as well as an address part i.e. next, which points to the next node in sequence of nodes. The operations we can perform on singly linked lists are insertion, deletion and traversal.

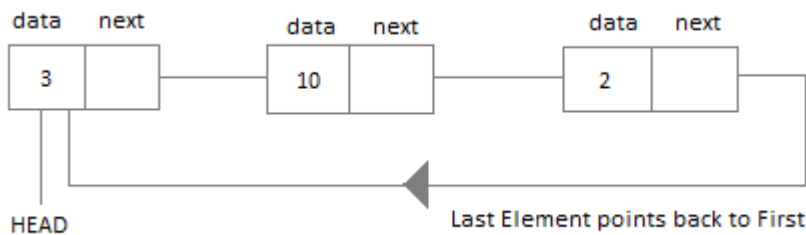




- **Doubly Linked List :** In a doubly linked list, each node contains two links the first link points to the previous node and the next link points to the next node in the sequence.



- **Circular Linked List :** In the circular linked list the last node of the list contains the address of the first node and forms a circular chain.



## Linear Linked List

The element can be inserted in linked list in 2 ways :

- Insertion at beginning of the list.
- Insertion at the end of the list.

We will also be adding some more useful methods like :

- Checking whether Linked List is empty or not.
- Searching any element in the Linked List
- Deleting a particular Node from the List

Before inserting the node in the list we will create a class **Node**. Like shown below :

```

class Node {
    public:
        int data;
        //pointer to the next node
        node* next;
    node() {
        data = 0;
        next = NULL;
    }
    node(int x) {
        data = x;
        next = NULL;
    }
}

```

We can also make the properties `data` and `next` as private, in that case we will need to add the getter and setter methods to access them. You can add the getters and setter like this :

```

int getData() {
    return data;
}
void setData(int x) {
    this.data = x;
}
node* getNext() {
    return next;
}
void setNext(node *n) {
    this.next = n;
}

```

Node class basically creates a node for the data which you enter to be included into Linked List. Once the node is created, we use various functions to fit in that node into the Linked List.

### Linked List class

As we are following the complete OOPS methodology, hence we will create a separate class for **Linked List**, which will have all its methods. Following will be the Linked List class :

```

class LinkedList {
    public:
        node *head;
        //declaring the functions

```

```

//function to add Node at front
int addAtFront(node *n);
//function to check whether Linked list is empty
int isEmpty();
//function to add Node at the End of list
int addAtEnd(node *n);
//function to search a value
node* search(int k);
//function to delete any Node
node* deleteNode(int x);

LinkedList() {
    head = NULL;
}
}

```

### Insertion at the Beginning

Steps to insert a Node at beginning :

1. The first Node is the Head for any Linked List.
2. When a new Linked List is instantiated, it just has the Head, which is Null.
3. Else, the Head holds the pointer to the first Node of the List.
4. When we want to add any Node at the front, we must make the head point to it.
5. And the Next pointer of the newly added Node, must point to the previous Head, whether it be NULL(in case of new List) or the pointer to the first Node of the List.
6. The previous Head Node is now the second Node of Linked List, because the new Node is added at the front.

```

int LinkedList :: addAtFront(node *n) {
    int i = 0;
    //making the next of the new Node point to Head
    n->next = head;
    //making the new Node as Head
    head = n;
    i++;
    //returning the position where Node is added
    return i;
}

```

### Inserting at the End

Steps to insert a Node at the end :

1. If the Linked List is empty then we simply, add the new Node as the Head of the Linked List.
2. If the Linked List is not empty then we find the last node, and make it' next to the new Node, hence making the new node the last Node.

```
int LinkedList :: addAtEnd(node *n) {  
//If list is empty  
    if(head == NULL) {  
//making the new Node as Head  
        head = n;  
//making the next pointe of the new Node as Null  
        n->next = NULL;  
    }  
    else {  
//getting the last node  
        node *n2 = getLastNode();  
        n2->next = n;  
    } }  
node* LinkedList :: getLastNode() {  
//creating a pointer pointing to Head  
    node* ptr = head;  
//Iterating over the list till the node whose Next pointer points to null  
//Return that node, because that will be the last node.  
    while(ptr->next!=NULL) {  
//if Next is not Null, take the pointer one step forward  
        ptr = ptr->next;  
    }  
    return ptr;  
}
```

### Searching for an Element in the List

In searhing we do not have to do much, we just need to traverse like we did while getting the last node, in this case we will also compare the **data** of the Node. If we get the Node with the same data, we will return it, otherwise we will make our pointer point the next Node, and so on.

```
node* LinkedList :: search(int x) {  
    node *ptr = head;  
    while(ptr != NULL && ptr->data != x) {  
//until we reach the end or we find a Node with data x, we keep moving  
        ptr = ptr->next;  
    }
```

```
}  
return ptr;  
}
```

## Deleting a Node from the List

Deleting a node can be done in many ways, like we first search the Node with **data** which we want to delete and then we delete it. In our approach, we will define a method which will take the **data** to be deleted as argument, will use the search method to locate it and will then remove the Node from the List.

To remove any Node from the list, we need to do the following :

- If the Node to be deleted is the first node, then simply set the Next pointer of the Head to point to the next element from the Node to be deleted.
- If the Node is in the middle somewhere, then find the Node before it, and make the Node before it point to the Node next to it.

```
node* LinkedList :: deleteNode(int x) {
```

```
//searching the Node with data x
```

```
node *n = search(x);
```

```
node *ptr = head;
```

```
if(ptr == n) {
```

```
    ptr->next = n->next;
```

```
    return n;
```

```
}
```

```
else {
```

```
    while(ptr->next != n) {
```

```
        ptr = ptr->next;
```

```
    }
```

```
    ptr->next = n->next;
```

```
    return n;
```

```
}
```

```
}
```

## Checking whether the List is empty or not

We just need to check whether the **Head** of the List is **NULL** or not.

```
int LinkedList :: isEmpty() {
```

```
    if(head == NULL) {
```

```
        return 1;
```

```
    }
```

```
    else { return 0; }
```

```
}
```

Now you know a lot about how to handle List, how to traverse it, how to search an element. You can yourself try to write new methods around the List.

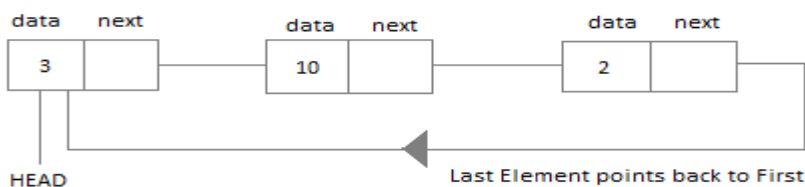
If you are still figuring out, how to call all these methods, then below is how your `main()` method will look like. As we have followed OOP standards, we will create the objects of **LinkedList** class to initialize our List and then we will create objects of **Node** class whenever we want to add any new node to the List.

```
int main() {
    LinkedList L;
    //We will ask value from user, read the value and add the value to our Node
    int x;
    cout <<"Please enter an integer value : ";
    cin >> x;
    Node *n1;
    //Creating a new node with data as x
    n1 = new Node(x);
    //Adding the node to the list
    L.addAtFront(n1);
}
```

Similarly you can call any of the functions of the LinkedList class, add as many Nodes you want to your List.

## Circular Linked List

Circular Linked List is little more complicated linked data structure. In the circular linked list we can insert elements anywhere in the list whereas in the array we cannot insert element anywhere in the list because it is in the contiguous memory. In the circular linked list the previous element stores the address of the next element and the last element stores the address of the starting element. The elements points to each other in a circular way which forms a circular chain. The circular linked list has a dynamic size which means the memory can be allocated when it is required.



## Application of Circular Linked List

- The real life application where the circular linked list is used is our Personal Computers, where multiple applications are running. All the running applications are kept in a circular linked list and the OS gives a fixed time slot to all for running. The Operating System keeps on iterating over the linked list until all the applications are completed.

- Another example can be Multiplayer games. All the Players are kept in a Circular Linked List and the pointer keeps on moving forward as a player's chance ends.
- Circular Linked List can also be used to create Circular Queue. In a Queue we have to keep two pointers, FRONT and REAR in memory all the time, where as in Circular Linked List, only one pointer is required.

### Implementing Circular Linked List

Implementing a circular linked list is very easy and almost similar to linear linked list implementation, with the only difference being that, in circular linked list the last **Node** will have it's **next** point to the **Head** of the List. In Linear linked list the last Node simply holds NULL in it's next pointer.

So this will be our Node class, as we have already studied in the lesson, it will be used to form the List.

```
class Node {
    public:
        int data;
        //pointer to the next node
        node* next;

    node() {
        data = 0;
        next = NULL;
    }

    node(int x) {
        data = x;
        next = NULL;
    }
}
```

### **Circular Linked List**

Circular Linked List class will be almost same as the Linked List class that we studied in the previous lesson, with a few difference in the implementation of class methods.

```
class CircularLinkedList {
    public:
        node *head;
        //declaring the functions
```

```

//function to add Node at front
int addAtFront(node *n);
//function to check whether Linked list is empty
int isEmpty();
//function to add Node at the End of list
int addAtEnd(node *n);
//function to search a value
node* search(int k);
//function to delete any Node
node* deleteNode(int x);

```

```

CircularLinkedList() {
    head = NULL;
}
}

```

### Insertion at the Beginning

Steps to insert a Node at beginning :

1. The first Node is the Head for any Linked List.
2. When a new Linked List is instantiated, it just has the Head, which is Null.
3. Else, the Head holds the pointer to the first Node of the List.
4. When we want to add any Node at the front, we must make the head point to it.
5. And the Next pointer of the newly added Node, must point to the previous Head, whether it be NULL(in case of new List) or the pointer to the first Node of the List.
6. The previous Head Node is now the second Node of Linked List, because the new Node is added at the front.

```

int CircularLinkedList :: addAtFront(node *n) {
    int i = 0;
    /* If the list is empty */
    if(head == NULL) {
        n->next = head;
    }
    //making the new Node as Head
    head = n;
    i++;
}
else {
    n->next = head;
}
//get the Last Node and make its next point to new Node

```



```

Node* last = getLastNode();
last->next = n;
//also make the head point to the new first Node
head = n;
i++;
}
//returning the position where Node is added
return i;
}

```

### Insertion at the End

Steps to insert a Node at the end :

1. If the Linked List is empty then we simply, add the new Node as the Head of the Linked List.
2. If the Linked List is not empty then we find the last node, and make it' next to the new Node, and make the next of the Newly added Node point to the Head of the List.

```

int CircularLinkedList :: addAtEnd(node *n) {
//If list is empty
if(head == NULL) {
//making the new Node as Head
head = n;
//making the next pointer of the new Node as Null
n->next = NULL;
}
else {
//getting the last node
node *last = getLastNode();
last->next = n;
//making the next pointer of new node point to head
n->next = head;
}
}

```

### Searching for an Element in the List

In searching we do not have to do much, we just need to traverse like we did while getting the last node, in this case we will also compare the **data** of the Node. If we get the Node with the same data, we will return it, otherwise we will make our pointer point the next Node, and so on.

```

node* CircularLinkedList :: search(int x) {
node *ptr = head;

```

```

while(ptr != NULL && ptr->data != x) {
    ptr = ptr->next;
}
return ptr;
}

```

### Deleting a Node from the List

Deleting a node can be done in many ways, like we first search the Node with **data** which we want to delete and then we delete it. In our approach, we will define a method which will take the **data** to be deleted as argument, will use the search method to locate it and will then remove the Node from the List.

To remove any Node from the list, we need to do the following :

- If the Node to be deleted is the first node, then simply set the Next pointer of the Head to point to the next element from the Node to be deleted. And update the next pointer of the Last Node as well.
- If the Node is in the middle somewhere, then find the Node before it, and make the Node before it point to the Node next to it.
- If the Node is at the end, then remove it and make the new last node point to the head.

```

node* CircularLinkedList :: deleteNode(int x) {
//searching the Node with data x
    node *n = search(x);
    node *ptr = head;
if(ptr == NULL) {
    cout <<"List is empty";
    return NULL;
} else if(ptr == n) {
    ptr->next = n->next;
    return n;
} else {
    while(ptr->next != n) {
        ptr = ptr->next;
    }
    ptr->next = n->next;
    return n;
}
}

```



## CH SRILAXMI

(Ph.D.), M.Tech

Prof. Srilakshmi Cherukuri working as an Assistant Professor & HoD in the CSE (AI&ML) Department at Narsimha Reddy Engineering College, Hyderabad. She secured a Master of Technology in CSE. She is pursuing a Ph.D., at GITAM University, Hyderabad, India. She has been in the field of teaching profession for more than 18 years. She has presented more than 25 papers in national and International Journals, Conferences, and Symposiums. Her main areas of interest include Deep Learning and Image Processing. Throughout her career, Ch Srilakshmi has been passionate about teaching and sharing her knowledge with others. She has conducted numerous workshops and seminars on programming languages, with a particular focus on C programming.



Your roots to success...

**NARSIMHA REDDY ENGINEERING COLLEGE**  
**UGC AUTONOMOUS INSTITUTION**

Maisammaguda (V), Kompally - 500100, Secunderabad, Telangana State, India

UGC - Autonomous Institute  
Accredited by NBA & NAAC with 'A' Grade  
Approved by AICTE  
Permanently affiliated to JNTUH