C language

Index

Chapter 1 Introduction to C Language

What is C ?

C is a programming language developed at AT & T's Bell Laboratories of USA in 1972.

It was designed send written by a man named Dennis Ritchie.

why bother to learn C today"

(a) I believe that nobody can learn C++ or Java directly.
(b) C language will strong the basic programming skills.
(c) Major parts of popular operating systems like Windows, UNIX, Linux is still written in C.
(d) Many popular gaming frameworks have been built using C language.
(e) At times one is required to very closely interact with the hardware devices.

Getting Started with C

Communicating with a computer involves speaking the language the computer understsends, which immediately rules out English as the language of communication with computer.

The C Character Set

A character denotes any alphabet, digit or special symbol used to represent information.

Constants, Variables send Keywords

Types of C Constants

C constants can be divided into two major categories:

(a) Primary Constants
(b) Secondary Constants

C Variables

Rules for Constructing Variable Names

(a) A variable name is any combination of 1 to 31 alphabets, digits or underscores.
(b) The first character in the variable name must be an alphabet or underscore
(c) No commas or blanks are allowed within a variable name.
(d) No special symbol other than an underscore (as in gross_sal) can be used in a variable name.

C Keywords

The keywords are also called 'Reserved words'.

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

Basic Program Structure

```
#include<stdioo.h>
#include<conio.h>
void maine(){
//variables
//functions
}
```

Compilation send Execution

- Once the program has been typed it needs to be converted to machine language (0s send 1s) before the machine can execute it. To carry out this conversion we need another program called Compiler.
- To type your C program you need another program called Editor.

Turbo C/C++

C Instructions

There are basically three types of instructions in C:

(a) ***Type Declaration Instruction*** − To declare the type of variables used in a C program.

Ex:- int bas ;
float rs, grosssal ;

(b) ***Arithmetic Instruction*** − To perform arithmetic operations between constants send variables.
Ex:- float data = alpha * beta / gamma + 3.2 * 2 / 5 ;

(c) ***Control Instruction*** − To control the sequence of execution of various statements in a C program.
For ex: if-else , loops, switch etc.

Integers send Float Conversions

Operation	Result	Operation	Result
5 / 2	2	2 / 5	0
5.0 / 2	2.5	2.0 / 5	0.4
5 / 2.0	2.5	2 / 5.0	0.4
5.0 / 2.0	2.5	2.0 / 5.0	0.4

Type Conversion in Assignments

Hierarchy of Operations

Priority	Operators	Description
1st	* / %	multiplication, division, modular division
2nd	+ -	addition, subtraction
3rd	=	assignment

Associativity of Operators

When an expression contains two operators of equal priority the tie between them is settled using the associativity of the operators.

Associativity can be of two types—Left to Right or Right to Left.

Unambiguous : It must not be involved in evaluation of any other sub-expression.

Operator	Left	Right	Remark
/	3	2 or 2 * 5	Left operand is unambiguous, Right is not
*	3 / 2 or 2	5	Right operand is unambiguous, Left is not

Consider the expression

a = 3 / 2 * 5 ; Here there is a tie between operators of same priority, that is between / send *.

Control Instructions in C

The 'Control Instructions' enable us to specify the order in which the various instructions in a program are to be executed by the computer.

They are:

(a) Sequence Control Instruction

(b) Selection or Decision Control Instruction

(c) Repetition or Loop Control Instruction

(d) **Case Control Instruction**

Questions and Answers

Q:1 Pick up the correct alternative for each of the following questions:

(a) C language has been developed by

(1) Ken Thompson

(2) Dennis Ritchie

(3) Peter Norton

(4) Martin Richards

(b) C can be used on

(1) Only MS-DOS operating system

(2) Only Linux operating system

(3) Only Windows operating system

(4) All the above

(c) C programs are converted into machine language with the help of

(1) An Editor

(2) A compiler

(3) An operating system

(4) None of the above

(d) The real constant in C can be expressed in which of the following forms

(1) Fractional form only

(2) Exponential form only

(3) ASCII form only

(4) Both fractional and exponential forms

(e) A character variable can at a time store

(1) 1 character

(2) 8 characters

(3) 254 characters

(4) None of the above

(f) The statement char ch = ‘Z’ would store in ch

(1) The character Z

(2) ASCII value of Z

(3) Z along with the single inverted commas

(4) Both (1) and (2)

(g) Which of the following is NOT a character constant

(1) ‘Thank You’

(2) ‘Enter values of P, N, R’

(3) ‘23.56E-03’

(4) All the above

(h) The maximum value that an integer constant can have is

(1) -32767

(2) 32767

(3) 1.7014e+38

(4) –1.7014e+38

(i) A C variable cannot start with

(1) An alphabet

(2) A number

(3) A special symbol other than underscore

(4) Both (2) & (3) above

(j) Which of the following statement is wrong

(1) mes = 123.56 ;

(2) con = 'T' * 'A' ;

(3) this = 'T' * 20 ;

(4) 3 + a = b ;

Q:2 Write C programs for the following (Coding):-

a) Ramesh’s basic salary is input through the keyboard. His dearness allowance is 40% of basic salary, and house rent allowance is 20% of basic salary. Write a program to calculate his gross salary.

b) Two numbers are input through the keyboard into two locations C and D. Write a program to interchange the contents of C and D.

c) If a ifve-digit number is input through the keyboard, write a program to calculate the sum of its digits. (Hint: Use the modulus operator ‘%’)

d) If a ifve-digit number is input through the keyboard, write a program to reverse the number.

e) If a four-digit number is input through the keyboard, write a program to obtain the sum of the ifrst and last digit of this number.

Chapter 2 The Operators

What are Operators in C?

Operators can be defined as the symbols that help us to perform specific mathematical, relational, bitwise, conditional, or logical computations on operands. In other words, we can say that an operator operates the operands.

Types of Operators in C

C has many built-in operators and can be classified into 5 types:

1. Arithmetic Operators
2. Relational Operators
3. Logical Operators
4. Bitwise Operators
5. Assignment Operators

Arithmetic Operations in C

These operators are used to perform arithmetic/mathematical operations on operands. Examples: (+, -, *, /, %,++,–).

Relational Operators in C

These are used for the comparison of the values of two operands. Examples: (>,<,<=,>=,==,!=)

Logical Operator in C

Logical Operators are used to combining two or more conditions/constraints or to complement the evaluation of the original condition in consideration. The result of the operation of a logical operator is a Boolean value either true or false. Examples: (&& , || , !)

Operands		Results			
x	y	!x	!y	x && y	x || y
0	0	1	1	0	0
0	non-zero	1	0	0	0
non-zero	0	0	1	0	1
non-zero	non-zero	0	0	1	1

Bitwise Operators in C

The Bitwise operators are used to perform bit-level operations on the operands. The operators are first converted to bit-level and then the calculation is performed on the operands. Examples: (& , | , ~ , ^ , >> , << , >>>)

Assignment Operators in C

Assignment operators are used to assign value to a variable. The left side operand of the assignment operator is a variable and the right side operand of the assignment operator is a value. Example: (= , += , -= , /=)

Hierarchy of Operators Revisited

Operators	Type
!	Logical NOT
* / %	Arithmetic and modulus
+ -	Arithmetic
< > <= >=	Relational
== !=	Relational
&&	Logical AND
||	Logical OR
=	Assignment

Questions and Answers

some questions related to operators in the C programming language:

1. What is an operator in the C language, and what is its primary purpose?

2. How are operators categorized in C? Can you name some categories?

3. Explain the difference between unary, binary, and ternary operators.

4. List some common arithmetic operators in C and provide examples of their use.

5. What is the modulus operator in C, and how is it used?

6. Describe the purpose and usage of the increment (++) and decrement (--) operators.

7. What are relational operators in C, and when are they typically used in programming?

8. Explain the logical AND (&&) and logical OR (||) operators in C with examples.

9. What is the purpose of the bitwise AND (&) and bitwise OR (|) operators in C?

10. Describe the NOT operator (~) in C and its application in bitwise operations.

11. How do you use the conditional operator (?:) in C, and what is its purpose?

12. What is the comma operator (,) in C, and where might it be useful?

13. How can you use the sizeof operator in C to determine the size of a data type or variable?

14. Explain the assignment operator (=) and its role in assigning values to variables.

15. What is operator precedence in C, and why is it important in evaluating expressions?

16. What is operator associativity, and how does it affect the order of operations in expressions?

17. Can you provide an example of operator overloading in C++? How is it different from C?

18. Discuss the role of the conditional operator (?:) in creating compact if-else statements.

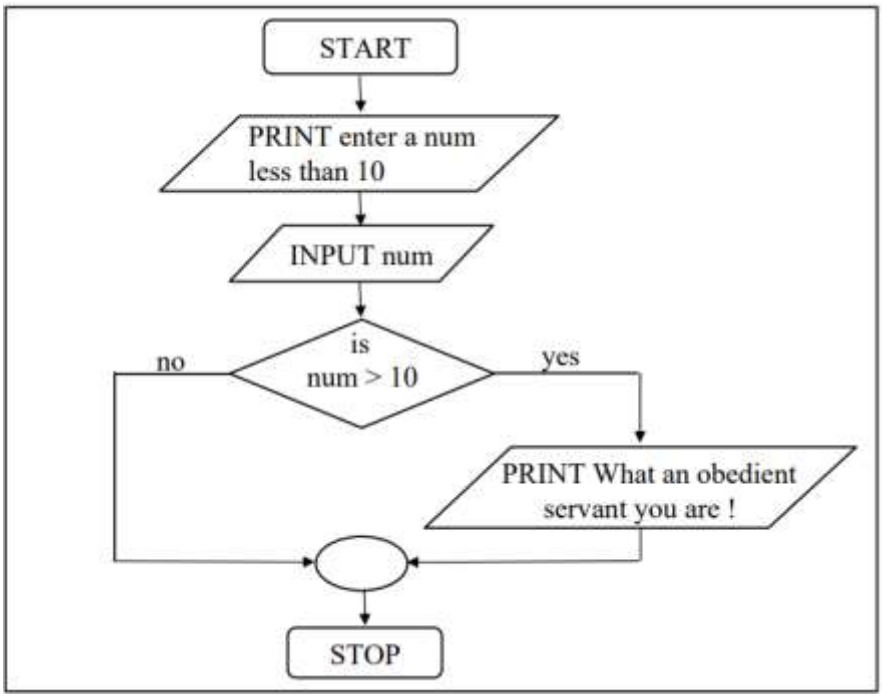
19. How do you use the shift operators (<< and >>) in C, and what are their common applications?

20. Explain the difference between prefix and postfix increment/decrement operators in C.

Chapter 3 The Decision Control Structure

A decision control instruction can be implemented in C using:

(a) The if statement
(b) The if-else statement
(c) The conditional operators



The if Statement

```
if ( this condition is true )
execute this statement ;
```

Multiple Statements within if

```
if ( this condition is true )
{
execute this statement ;
}
```

The if-else Statement

```
if ( this condition is true )
{
execute this statement ;
}
else
{
execute this statement if condition false ;
}
```

Nested if-elses

```
if ( this condition is true )
{
execute this statement ;
}
else
{
if(condition){
execute this statement if condition false ;
}
}
```

The else if Clause

```
if ( this condition is true )
execute this statement ;
else  if(condition){
execute this statement if condition false ;
}
```

The Conditional Operators

The conditional operators ? send : are sometimes called ternary operators since they take three arguments.

```
expression 1 ? expression 2 : expression 3
```

This is also known as short-hsend of if else, which is known as the **ternary operator** because it consists of three opersends. It can be used to replace multiple lines of code with a single line. It is often used to replace simple if else statements

Questions and Answers

1. What is the purpose of the if statement in C?
2. How does the if-else statement work in C, and what is its syntax?
3. Can you explain the difference between an if statement and an if-else statement?
4. What happens when the condition in an if statement is true, and what happens when it is false?
5. How can you use multiple conditions in a single if-else statement?
6. What are the advantages of using if-else statements in C?
7. What is the purpose of the nested if-else statement in C?
8. How do you write a nested if-else statement in C, and what is its syntax?
9. Can you provide an example of a nested if-else statement to solve a practical problem?
10. How can you use logical operators (e.g., && and ||) within if-else statements?
11. What are the potential issues or pitfalls to watch out for when using nested if-else statements?
12. Explain the concept of the "else if" statement in C. How is it used, and why is it helpful?
13. How can you handle situations where you have multiple conditions with different outcomes in a nested if-else structure?
14. When is it appropriate to use a switch statement instead of nested if-else statements?
15. What is the role of the "else" part in an if-else statement, and how does it affect the program's flow?
16. Write a C program that accepts three integers as input and uses if-else statements to find and print the largest of the three numbers.
17. Create a C program that takes a year as input and checks whether it is a leap year or not using an if-else statement. Display the result.

18. Write a C program that reads a character from the user and uses an if-else statement to check if it is a vowel or a consonant. Display the result.

19. Create a C program that takes two numbers as input and uses an if-else statement to determine and print whether the first number is a multiple of the second number.

20. Write a C program that reads a positive integer as input and uses an if-else statement to check if it is a prime number. Display the result.

21. Create a C program that calculates the cost of a product based on its price and a discount percentage. Use an if-else statement to apply the discount if the price is above a certain threshold.

22. Write a C program that accepts a temperature in Celsius and uses an if-else statement to convert it to Fahrenheit if the user requests it, or to Kelvin if not. Display the result.

23. Create a C program that reads three angles of a triangle as input and uses an if-else statement to determine and print whether the triangle is equilateral, isosceles, or scalene.

Chapter 4 The Loop Control Structure

Loops

The versatility of the computer lies in its ability to perform a set of instructions repeatedly. This involves repeating some portion of the program either a specified number of times or until a particular condition is being satisfied.

They are:

(a) Using a for statement

The for allows us to specify three things about a loop in a single line:

- ✓ Setting a loop counter to an iintial value.
- ✓ Testing the loop counter to determine whether its value has reached the number of repetitions desired.
- ✓ Increasing the value of loop counter each time the program segment within the loop has been executed.

```
for( iintialize counter ; test counter ; increment counter ) {
/* code */
}
```

(b) Using a while statement

```
Iintialize loop counter;
while( this condition is true ) {
/* code */
increment loop counter ;
}
```

(c) Using a do-while statement

Note that this loop ensures that statements within it are executed at least once

```
Iintialize loop counter;
do{
/* code */
increment loop counter ;
} while(  condition  );
```

Nesting of Loops

```
do{
do{
/* code */
} while(  condition  );
} while(  condition  );
```

The **break** Statement

We often come across situations where we want to jump out of a loop instantly, without waiting to get back to the conditional test. The keyword break allows us to do this.

A break is usually associated with an if.

The **continue** Statement

When continue is encountered inside any loop, control automatically passes to the beginning of the loop.

A continue is usually associated with an if.

Questions and Answers

1. What is a loop in C, and why is it used?
2. What are the three primary types of loops in C?
3. How do you declare an infiinte loop in C?
4. What is the difference between a "while" loop and a "do - while" loop?
5. Explain the "for" loop structure in C. What are the components of a "for" loop?
6. How can you exit a loop prematurely in C?
7. What is the purpose of the "break" statement in a loop?
8. How is the "continue" statement used in a loop?
9. What is a nested loop, and why would you use one?
10. How do you iterate through an array using a loop in C?
11. Explain the concept of a "loop control variable" or "loop counter."
12. What is the significance of the loop condition in a "while" or "for" loop?
13. How can you iterate through a linked list using a loop in C?
14. What is the purpose of the "goto" statement, and how can it be used to control loops?
15. How can you determine the number of iterations a loop will execute in advance?
16. What is the difference between a "pre - test" loop and a "post - test" loop?
17. How can you create a loop that counts in reverse order, such as from 10 down to 1?
18. Explain the "nested loop" concept and provide an example.
19. What is the purpose of a "loop label" in C, and how is it used?
20. Can you provide an example of an application where an "infiinte loop" is useful in C?

Chapter 5 The Case Control Structure

Decisions Using switch

The control statement that allows us to make a decision from the number of choices is called a switch.

switch-case-default, since these three keywords go together to make up the control statement.

```
switch ( integer expression ) {
case constant 1 : do this ;
                                  break;
case constant 2 : do this ;
                                 break;
case constant 3 : do this ;
                                  break;
default : do this ;
}
```

- ✓ The integer expression following the keyword ***switch*** is any C expression that will yield an integer value.
- ✓ The keyword ***case*** is followed by an integer or a character constant.
- ✓ If no match is found with any of the case statements, only the statements following the ***default*** are executed.
- ✓ it is upto you to get out of the switch then send there by using a break statement.
- ✓ Note that there is no need for a break statement after the default, since the control comes out of the switch anyway.

switch Versus if-else Ladder

There are some things that you simply cannot do with a switch. These are:

(a) A float expression cannot be tested using a switch

(b) Cases can never have variable expressions (for example it is wrong to say case a +3 :)

(c) Multiple cases cannot use same expressions. Thus the following switch is illegal:

The goto Keyword

Avoid goto keyword! They make a C programmer's life miserable.

The big problem with gotos is that when we do use them we can never be sure how we got to a certain point in our code.

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

void main( )
{
 int goals ;
 printf ( "Enter the number of goals scored against India" ) ;
 scanf ( "%d", &goals ) ;
 sos :
 printf("sos runnig");
 if ( goals <= 5 )
 goto sos ;
 else
 {
 printf ( "About time soccer players learnt C\n" ) ;
 printf ( "send said goodbye! adieu! to soccer" ) ;
 }

 printf ( "To err is human!" ) ;
}
```

A few remarks about the program would make the things clearer.

- If the condition is satisifed the goto statement transfers control to the label 'sos', causing printf( ) following sos to be executed.
- The label can be on a separate line or on the same line as the statement following it, as in, sos : printf ("To err is human!") ;
- Any number of gotos can take the control to the same label.
- The exit( ) function is a stsendard library function which terminates the execution of the program. It is necessary to use

this function since we don't want the statement printf ("To err is human!") to get executed after execution of the else block.

Questions and Answers

1. What is the purpose of the "switch" statement in C, and how does it differ from "if - else" statements?
2. Can you explain the syntax of the "switch" statement in C? What are the essential components of a "switch" statement?
3. How does the "switch" statement work in terms of control flow? Describe the sequence of execution when a "switch" statement is encountered.
4. What is the role of "case" labels within a "switch" statement, and how are they used to control program flow?
5. What is the purpose of the "default" case in a "switch" statement, and when is it executed?
6. Can a "switch" statement be used with data types other than integers, such as characters or enums? If so, how?
7. What happens if you forget to include a "break" statement in one of the "case" labels in a "switch" statement? How does this affect program execution?
8. Discuss scenarios where using a "switch" statement is more appropriate than using a series of "if - else" statements.
9. Write a C program that uses a "switch" statement to take an integer input (1 - 7) representing a day of the week and prints the corresponding day's name.
10. Write a C program that uses a "switch" statement to implement a simple calculator. The program should take two numbers and an operator (+, -, *, /) as input and perform the corresponding calculation.
11. How does the "switch" statement handle fall - through behavior? Provide an example of fall - through and explain its implications.
12. Write a C program that uses a "switch" statement to convert a numerical grade (0 - 100) into a letter grade (A, B, C, etc.) according to a typical grading scale.

Chapter 6 The Functions & Pointers

What is a Function

A function is a self-contained block of statements that perform a coherent(specific) task of some kind. Send It will be used again send again in program when we need.

- *Function has a name , return type and arguments.*
- *Function is a way to achieve modularization*
- *Function are Pre-deifned send user-deifned*
- *Predeifned functions are declared in header files send deifned in library files.*

Example:-

```
#include<stdio.h>
#include<conio.h>
void sum(int,int); //function declaration gllobly
void main()
{
  //variable declaration send assign
  int a=5;
  int b=8;

  //function call
  sum(a,b);

}
//funciton deifne
void sum(int x,int y)
{
    int c;
    c = x+y;
    printf("%d",c);
  }
```

syntax

```
returnType FunctionName(){
    //code...
}
```

Why Use Functions

(a) Writing functions avoids rewriting the same code over send over.

(b) Using functions it becomes easier to write programs send keep track of what they are doing.

Deifintion, Declaration and Call


```
#include<stdio.h>
void main( )
{
  void fun( );
  printf("You are in main");
  fun( );
}
void fun( )
{
 printf("You are in fun");
}
```

Declaration of printf send scanf

Function Declaration

Function Call

Function Deifintion

Declaration

- ❖ Function Declaration is also known as Function prototype
- ❖ Function need to be declared before use (just like variables)
- ❖ Function can be declared locally or globally
- ❖ Return Type functionName(argumentList);
- ❖ Function deifintion is block of code

Types of Formal Arguments

- ✓ Formal arguments can be two types
  - Ordinary variables of any type
  - Pointer variables

Beneifts of function

- ✓ Easy to read
- ✓ Easy to Modify
- ✓ Avoids rewriting of same code
- ✓ Easy to debug
- ✓ Better memory utilization

Function Saves memory

- ❖ Function in a program is to save memory space which becomes appreciable when a function is likely to be called many times.

Function is time consuming

- However every time a function is called, it takes lot of extra time in executing a series of instructions for tasks such as jumping to the functions, saving registers, pushing arguments into the stack send returning to the calling function.

So…

So when function is small it is worthless to spend so much extra time in such tasks in cost of saving comparatively small space.

Questions and Answers

Functions:

1. What is a function in C, and why are they important in programming?
2. How do you declare a function in C?
3. What is the difference between a function declaration and a function definition?
4. What is a function prototype, and why is it used?
5. Explain the concept of function parameters in C.
6. What is the purpose of the "return" statement in a function?
7. How can you call a function in C, and what is the syntax for function calls?
8. What is a recursive function in C, and can you provide an example?
9. What is the difference between call by value and call by reference in function parameter passing?
10. How can you pass an array to a function in C?
11. What is the "main" function in C, and why is it special?
12. What is a function pointer, and how is it declared in C?

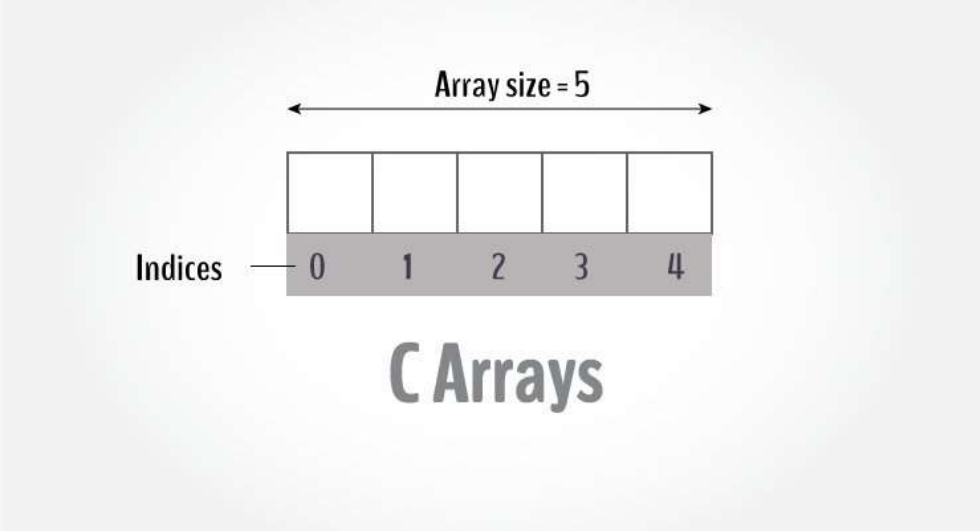
Pointers:

1. What is a pointer in C, and why are they important?

2. How do you declare a pointer variable in C?
3. Explain the concept of pointer arithmetic in C.
4. What is the difference between a pointer and an array in C?
5. How do you use pointers to pass values between functions?
6. What is a NULL pointer, and why is it used?
7. How can you dynamically allocate memory using pointers in C?
8. What is the difference between the " - >" operator and the "." operator when working with structures and pointers?
9. How can you create an array of pointers in C?
10. Explain the concept of a double pointer (pointer to a pointer) and its use cases.
11. How do you deallocate memory when using dynamic memory allocation with pointers in C?
12. What is the significance of the "const" keyword when working with pointers?

Chapter 7 Array

Arrays in C



What is Array in C?

An array in C is a fixed-size collection of similar data items stored in contiguous memory locations.

Example of Array Declaration

```c
// C Program to illustrate the array declaration
#include <stdio.h>

int main()
{

    // declaring array of integers
    int arr_int[5];
    // declaring array of characters
    char arr_char[5];

    return 0;
```

```
}
```

Access Array Elements

We can access any element of an array in C using the array subscript operator **[ ]** and the index value *i* of the element.

```
array_name [index];
```

Types of Array in C

There are two types of arrays based on the number of dimensions it has. They are as follows:

1. One Dimensional Arrays (1D Array)
2. Multidimensional Arrays

Example of 1D Array in C


```c
// C Program to illustrate the use of 1D array
#include <stdio.h>

int main()
{

    // 1d array declaration
    int arr[5];

    // 1d array initialization using for loop
    for (int i = 0; i < 5; i++) {
        arr[i] = i * i - 2 * i + 1;
    }

    printf("Elements of Array: ");
    // printing 1d array by traversing using for loop
    for (int i = 0; i < 5; i++) {
        printf("%d ", arr[i]);
```

```
    }

    return 0;
}
```

Properties of Arrays in C

It is very important to understand the properties of the C array so that we can avoid bugs while using it.

The following are the main properties of an array in C:

1. Fixed Size

The array in C is a fixed-size collection of elements. The size of the array must be known at the compile time and it cannot be changed once it is declared.

2. Homogeneous Collection

We can only store one type of element in an array. There is no restriction on the number of elements but the type of all of these elements must be the same.

3. Indexing in Array

The array index always starts with 0 in C language. It means that the index of the first element of the array will be 0 and the last element will be N – 1.

4. Dimensions of an Array

A dimension of an array is the number of indexes required to refer to an element in the array. It is the number of directions in which you can grow the array size.

5. Contiguous Storage

All the elements in the array are stored continuously one after another in the memory. It is one of the defining properties of the array in C which is also the reason why random access is possible in the array.

6. Random Access

The array in C provides random access to its element i.e we can get to a random element at any index of the array in constant time complexity just by using its index number.

7. No Index Out of Bounds Checking

There is no index out-of-bounds checking in C/C++, for example, the following program compiles fine but may produce unexpected output when run.

Questions and Answers

1. What is an array in C, and how is it different from a simple variable?
2. How do you declare an array in C? Provide an example.
3. Explain the concept of array indexing. How are array elements accessed?
4. What is the maximum number of elements an array can hold in C?
5. Can the elements of an array have different data types in C?
6. What is the significance of the size of an array in terms of memory allocation?

7. How can you initialize an array in C? Provide examples of both one-dimensional and multi-dimensional arrays.
8. What is the difference between a one-dimensional array and a multi-dimensional array in C?
9. How do you find the length or size of an array in C?
10. Explain the process of passing an array to a function in C.
11. How can you sort an array in C? Discuss different sorting algorithms.
12. What are the common operations you can perform on arrays in C?
13. What is a dynamic array, and how is it different from a static array in C?
14. How do you access the last element of an array without knowing its size in advance?
15. What is the purpose of a sentinel value in array processing, and how is it used?
16. How do you initialize an array with all elements set to a specific value in C?
17. Can you have an array of structures in C? How is it defined and accessed?
18. Explain the concept of a jagged array in C.
19. How is memory allocated and deallocated for arrays in C?
20. What is the difference between an array and a pointer in C, and when would you use one over the other?

Chapter 8 Strings

What is String

A String in C programming is a sequence of characters terminated with a null character '\0'.

C String Declaration Syntax

Declaring a string in C is as simple as declaring a one-dimensional array. Below is the basic syntax for declaring a string.

```
char string_name[size];
```

C String Example

```
// C program to illustrate strings

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

int main()
{
    // declare and initialize string
    char str[] = "Sanjeev Sir";

    // print string
    printf("%s\n", str);

    int length = 0;
    length = strlen(str);

    // displaying the length of string
    printf("Length of string str is %d", length);

    return 0;
}
```

Standard C Library – String.h Functions

The C language comes bundled with **<string.h>** which contains some useful string-handling functions. Some of them are as follows:

Function Name	Description
strlen(string_name)	Returns the length of string name.
strcpy(s1, s2)	Copies the contents of string s2 to string s1.
strcmp(str1, str2)	Compares the first string with the second string. If strings are the same it returns 0.
strcat(s1, s2)	Concat s1 string with s2 string and the result is stored in the first string.
strlwr()	Converts string to lowercase.
strupr()	Converts string to uppercase.
strstr(s1, s2)	Find the first occurrence of s2 in s1.

C String Initialization

A string in C can be initialized in different ways. We will explain this with the help of an example. Below are the examples to declare a string with the name str and initialize it with "Sanjeev Sir".

Ways to Initialize a String in C

We can initialize a C string in 4 different ways which are as follows:

1. Assigning a string literal without size

String literals can be assigned without size. Here, the name of the string str acts as a pointer because it is an array.

```
char str[] = "Sanjeev Sir";
```

2. Assigning a string literal with a predefined size

String literals can be assigned with a predefined size. But we should always account for one extra space which will be assigned to the null character. If we want to store a string of size n then we should always declare a string with a size equal to or greater than n+1.

```
char str[50] = "Sanjeev Sir";
```

3. Assigning character by character with size

We can also assign a string character by character. But we should remember to set the end character as '\0' which is a null character.

```
char str[14] = { 'S','A','N','J','E','E','V','S','I','R','\0'};
```

4. Assigning character by character without size

We can assign character by character without size with the NULL character at the end. The size of the string is determined by the compiler automatically.

```
char str[] = { 'S','A','N','J','E','E','V','S','I','R','\0'};
```

Note: When a Sequence of characters enclosed in the double quotation marks is encountered by the compiler, a null character '\0' is appended at the end of the string by default.

Questions and Answers

1. How do you declare and initialize a string in C?
2. What is the difference between a character array and a string in C?

3. How can you find the length of a string in C?
4. Explain the **strlen()** function in C and its usage.
5. How can you copy one string to another in C?
6. What is a null-terminated string in C, and why is it important?
7. What are the standard library functions for comparing two strings in C?
8. How can you concatenate two strings in C?
9. Explain the concept of character-by-character traversal of a string in C.
10. How do you convert a string to an integer in C?
11. What is the purpose of the **strtok()** function in C, and how is it used?
12. How can you reverse a string in C?
13. What is the **strncpy()** function in C, and when is it useful?
14. What are the potential issues with buffer overflow when working with strings in C?
15. How can you check if a substring exists within a larger string in C?
16. Explain the use of the **sprintf()** function in C for string formatting.
17. How do you read a line of text from the standard input in C?
18. What is the difference between a constant string and a mutable string in C?
19. How can you convert a string to uppercase or lowercase in C?
20. What are the memory allocation functions for working with strings dynamically in C?

Chapter 9 Structures

C Structures

The structure in C is a user-defined data type that can be used to group items of possibly different types into a single type.

A structure contains a number of data types grouped together. These data types may or may not be of the same type.

Declaring a Structure

```
struct book {
char  name ;
float  price ;
int  pages ;
} ;
```


syntax

```
struct <structure name> {
structure element 1 ;
structure element 2 ;
structure element 3 ;
......  ......
} ;
```

Note the following points while declaring a structure type:

a) The closing brace in the structure type declaration must be
followed by a semicolon.

b) It is important to understsend that a structure type declaration
does not tell the compiler to reserve any space in memory.
c) All a structure declaration does is, it deifnes the 'form' of the
structure.
d) Usually structure type declaration appears at the top of the
source code file, before any variables or functions are deifned.

C Structure Definition

To use structure in our program, we have to define its instance. We can do that by creating variables of the structure type.

1. Structure Variable Declaration with Structure Template

```
struct structure_name {
    data_type member_name1;
    data_type member_name1;
    ....
    ....
}variable1, varaible2, ...;
```

2. Structure Variable Declaration after Structure Template

```
// structure declared beforehand
struct structure_name variable1, variable2, .......;
```

Access Structure Members

We can access structure members by using the (.) dot operator.

Syntax

```
structure_name.member1;
```

```
structure_name.member2;
```

Example of Structure in C

```c
// C program to illustrate the use of structures
#include <stdio.h>

// declaring structure with name str1
struct str1 {
    int i;
    char c;
    float f;
    char s[30];
};

// declaring structure with name str2
struct str2 {
    int ii;
    char cc;
    float ff;
} var; // variable declaration with structure template

// Driver code
int main()
{
    // variable declaration after structure template
    // initialization with initializer list and designated
    // initializer list
    struct str1 var1 = { 1, 'A', 1.00, "SanjeevSir" },
                var2;
    struct str2 var3 = { .ff = 5.00, .ii = 5, .cc = 'a' };

    // copying structure using assignment operator
    var2 = var1;

    printf("Struct 1:\n\ti = %d, c = %c, f = %f, s = %s\n",
        var1.i, var1.c, var1.f, var1.s);
```

```
    printf("Struct 2:\n\ti = %d, c = %c, f = %f, s = %s\n",
        var2.i, var2.c, var2.f, var2.s);
    printf("Struct 3\n\ti = %d, c = %c, f = %f\n", var3.ii,
        var3.cc, var3.ff);

    return 0;
}
```

typedef for Structures

The *typedef* keyword is used to define an alias for the already existing datatype.

```
// C Program to illustrate the use of typedef with
// structures
#include <stdio.h>

// defining structure
struct str1 {
    int a;
};

// defining new name for str1
typedef struct str1 str1;

// another way of using typedef with structures
typedef struct str2 {
    int x;
} str2;

int main()
{
    // creating structure variables using new names
    str1 var1 = { 20 };
    str2 var2 = { 314 };

    printf("var1.a = %d\n", var1.a);
    printf("var2.x = %d", var2.x);
```

```
    return 0;
}
```

Questions and Answers

1. What is a structure in C, and how is it defined?
2. How can you access individual members (fields) of a structure in C?
3. Explain the difference between a structure and a union in C.
4. What is the size of a structure in C, and how is it determined?
5. How can you initialize a structure in C?
6. What is a nested structure in C, and why would you use one?
7. Describe the concept of padding in C structures. Why is it necessary?
8. How can you pass a structure to a function in C?
9. What is a pointer to a structure, and how is it different from a structure variable?
10. Write a C program that defines a structure to represent a point in 2D space (x, y) and calculates the distance between two points.
11. Create a structure in C to represent a book with attributes like title, author, and price. Write a program that initializes and displays the details of a book.
12. Define a structure representing a student with attributes like name, roll number, and marks. Write a program that calculates the average marks of a group of students.
13. Explain the concept of a self-referential structure. Provide an example.
14. What is the difference between an array of structures and an array of pointers to structures in C?
15. Write a C program that uses a structure to implement a simple calculator, allowing the user to perform addition, subtraction, multiplication, and division.
16. How do you use the **typedef** keyword with structures in C, and what is its purpose?
17. What is the scope of a structure in C? Can you have global and local structures?

18. Write a C program that reads data for multiple employees (name, employee ID, salary) using an array of structures and finds the employee with the highest salary.
19. What is the role of bit fields in C structures? Provide an example.
20. Create a C program that uses structures to represent a date (day, month, year) and calculates the next day's date.

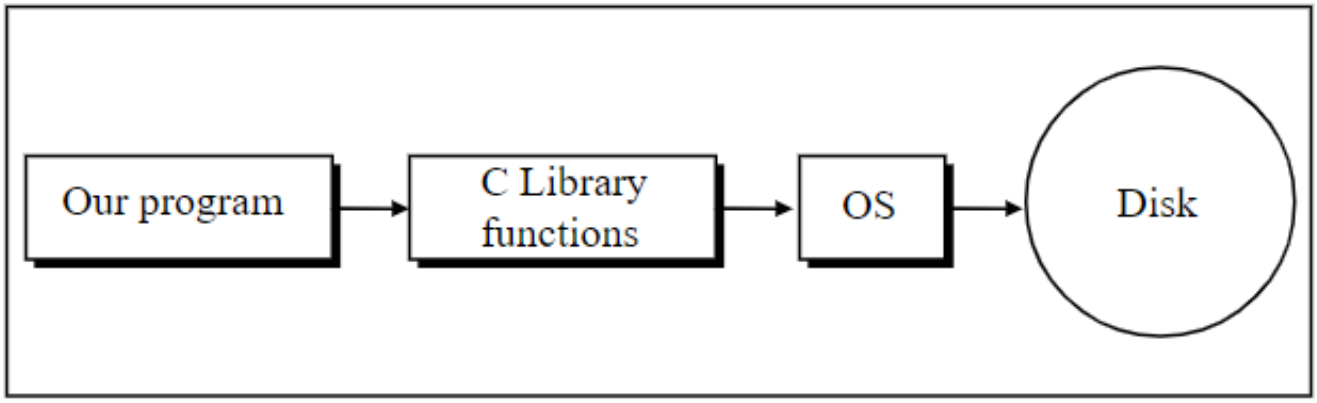
Chapter 10 File Handling

Memory is volatile send its contents would be lost once the program is terminated.

At such times it becomes necessary to store the data in a manner that can be later retrieved send displayed either in part or in whole. This medium is usually a 'file' on the disk.

Data Organization

All data stored on the disk is in binary form.



File Operations

There are different operations that can be carried out on a file. These are:

1. Creation of a new file
2. Opening an existing file
3. Reading from a file
4. Writing to a file
5. Closing a file

Types of Files in C

A file can be classified into two types based on the way the file stores the data. They are as follows:

- **Text Files**
- **Binary Files**

File Pointer in C

A file pointer is a reference to a particular position in the opened file. It is used in file handling to perform all file operations such as read, write, close, etc.

Syntax of File Pointer

```
FILE* pointer_name;
```

Open a File in C

For opening a file in C, the fopen() function is used with the filename or file path along with the required access modes.

Syntax of fopen()

```
FILE* fopen(const char *file_name, const char *access_mode);
```

File opening modes in C

File opening modes or access modes specify the allowed operations on the file to be opened. They are passed as an argument to the fopen() function.

Some of the commonly used file access modes are listed below:

Opening Modes	Description
r	Searches file. If the file is opened successfully fopen( ) loads it into memory and sets up a pointer that points to the

	first character in it. If the file cannot be opened fopen( ) returns NULL.
rb	Open for reading in binary mode. If the file does not exist, fopen( ) returns NULL.
w	Open for writing in text mode. If the file exists, its contents are overwritten. If the file doesn't exist, a new file is created. Returns NULL, if unable to open the file.
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	Searches file. If the file is opened successfully fopen( ) loads it into memory and sets up a pointer that points to the last character in it. It opens only in the append mode. If the file doesn't exist, a new file is created. Returns NULL, if unable to open the file.
ab	Open for append in binary mode. Data is added to the end of the file. If the file does not exist, it will be created.
r+	Searches file. It is opened successfully fopen( ) loads it into memory and sets up a pointer that points to the first character in it. Returns NULL, if unable to open the file.
rb+	Open for both reading and writing in binary mode. If the file does not exist, fopen( ) returns NULL.
w+	Searches file. If the file exists, its contents are overwritten. If the file doesn't exist a new file is created. Returns NULL, if unable to open the file.
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+	Searches file. If the file is opened successfully fopen( ) loads it into memory and sets up a pointer that points to the

	last character in it. It opens the file in both reading and append mode. If the file doesn't exist, a new file is created. Returns NULL, if unable to open the file.
ab+	Open for both reading and appending in binary mode. If the file does not exist, it will be created.

Create a File in C

The fopen() function can not only open a file but also can create a file if it does not exist already.

```
FILE *fptr;
fptr = fopen("filename.txt", "w");
```

Reading From a File

The file read operation in C can be performed using functions fscanf() or fgets().

Example:

```
FILE * fptr;
fptr = fopen("fileName.txt", "r");
fscanf(fptr, "%s %s %s %d", str1, str2, str3, &year);
char c = fgetc(fptr);
```

Write to a File

The file write operations can be performed by the functions fprintf() and fputs() with similarities to read operations.

Example:

```
FILE *fptr ;
fptr = fopen("fileName.txt", "w");
fprintf(fptr, "%s %s %s %d", "We", "are", "in", 2012);
fputc("a", fptr);
```

Closing a File

The fclose() function is used to close the file.

Example:

```
FILE *fptr ;
fptr= fopen(“fileName.txt”, “w”);
//---------- Some file Operations -------
fclose(fptr);
```

Example 1: Program to Create a File, Write in it, And Close the File

```
// C program to Open a File,
// Write in it, And Close the File
#include <stdio.h>
#include <string.h>

int main()
{

    // Declare the file pointer
    FILE* filePointer;

    // Get the data to be written in file
    char dataToBeWritten[50] = "GeeksforGeeks-A Computer "
                               "Science Portal for Geeks";

    // Open the existing file GfgTest.c using fopen()
    // in write mode using "w" attribute
    filePointer = fopen("GfgTest.c", "w");

    // Check if this filePointer is null
    // which maybe if the file does not exist
    if (filePointer == NULL) {
        printf("GfgTest.c file failed to open.");
```

```
    }
    else {

        printf("The file is now opened.\n");

        // Write the dataToBeWritten into the file
        if (strlen(dataToBeWritten) > 0) {

            // writing in the file using fputs()
            fputs(dataToBeWritten, filePointer);
            fputs("\n", filePointer);
        }

        // Closing the file using fclose()
        fclose(filePointer);

        printf("Data successfully written in file "
            "GfgTest.c\n");
        printf("The file is now closed.");
    }

    return 0;
}
```

Questions and Answers

1. How do you open a file in C for reading and writing, and what are the modes used for these operations?
2. What are the differences between text mode and binary mode when opening a file in C?
3. How can you check if a file has been successfully opened in C, and what function do you use to do so?
4. Explain the purpose of the **fopen()** function in C, and provide an example of its usage.
5. What is the difference between the **fprintf()** and **fscanf()** functions in C when working with files?

6. How do you close a file in C, and why is it important to close files after using them?
7. What is the purpose of the **feof()** and **ferror()** functions in C when dealing with file I/O?
8. How can you read a line of text from a file in C, and what function do you use for this task?
9. What is file positioning in C, and how do you set the file position indicator to a specific location within a file?
10. Explain the concepts of buffering and flushing when working with files in C.
11. How do you handle errors during file I/O operations in C, and what functions can be used to detect and handle these errors?
12. What is the purpose of the **rewind()** and **fseek()** functions in C when it comes to file positioning?
13. How can you create a new file or overwrite an existing file in C using file handling functions?
14. What is the difference between reading and writing data from/to a file using character-oriented and block-oriented functions in C?
15. What are the advantages and disadvantages of using binary files over text files in C for data storage?
16. How can you check the end-of-file condition while reading from a file in C?
17. Explain the difference between the **putc()** and **fputc()** functions in C for writing characters to a file.
18. What is a temporary file, and how can you create and work with temporary files in C?
19. How do you append data to the end of an existing file in C without overwriting its contents?
20. Can you explain the role of file permissions and access control when working with files in C?

Chapter 11 C Preprocessor

It is a program that processes our source program before it is passed to the compiler.

A C Preprocessor is just a text substitution tool send it instructs the compiler to do required pre-processing before the actual compilation. We'll refer to the C Preprocessor as CPP.

The following table lists all the preprocessor directives in C:

Preprocessor Directives	Description
#define	Used to define a macro
#undef	Used to undefine a macro
#include	Used to include a file in the source code program
#ifdef	Used to include a section of code if a certain macro is defined by #define
#ifndef	Used to include a section of code if a certain macro is not defined by #define
#if	Check for the specified condition
#else	Alternate code that executes when #if fails
#endif	Used to mark the end of #if, #ifdef, and #ifndef

These preprocessors can be classified based on the type of function they perform.

Types of C/C++ Preprocessors

There are 4 Main Types of Preprocessor Directives:

1. Macros
2. File Inclusion

3. Conditional Compilation
4. Other directives

1. Macros

In C, Macros are pieces of code in a program that is given some name. Whenever this name is encountered by the compiler, the compiler replaces the name with the actual piece of code.

Predefined Macros

```c
#include <stdio.h>

int main() {

   printf("File :%s\n", __FILE__ );
   printf("Date :%s\n", __DATE__ );
   printf("Time :%s\n", __TIME__ );
   printf("Line :%d\n", __LINE__ );
   printf("ANSI :%d\n", __STDC__ );

}
```

Example of Macro

```c
// C Program to illustrate the macro
#include <stdio.h>

// macro definition
#define LIMIT 5

int main()
{
    for (int i = 0; i < LIMIT; i++) {
        printf("%d \n", i);
    }

    return 0;
```

```
}
```

Macros With Arguments

We can also pass arguments to macros. Macros defined with arguments work similarly to functions.

Example

```
#define foo(a, b) a + b
#define func(r) r * r
```

2. File Inclusion

This type of preprocessor directive tells the compiler to include a file in the source code program.

Syntax

```
#include <file_name>
```

3. Conditional Compilation

There are the following preprocessor directives that are used to insert conditional code:

1. #if
2. #ifdef
3. #ifndef
4. #else
5. #elif
6. #endif

#endif directive is used to close off the #if, #ifdef, and #ifndef opening directives which means the preprocessing of these directives is completed.

Syntax

```
#ifdef macro_name
    statement1;
```

```
    statement2;
    statement3;
    .
    .
    .
    statementN;
#endif
```

4. Other Directives

Apart from the above directives, there are two more directives that are not commonly used. These are:

1. **#undef Directive**
2. **#pragma Directive**

Example:

```
// C program to illustrate the #pragma exit and pragma
// startup
#include <stdio.h>

void func1();
void func2();

// specifying funct1 to execute at start
#pragma startup func1
// specifying funct2 to execute before end
#pragma exit func2

void func1() { printf("Inside func1()\n"); }

void func2() { printf("Inside func2()\n"); }

// driver code
int main()
```

```c
{
    void func1();
    void func2();
    printf("Inside main()\n");

    return 0;
}
```

Questions and Answers

1. What is the C preprocessor, and what is its primary purpose in C?
2. How do you include header files in C using the preprocessor directive?
3. What is the difference between **#include <header.h>** and **#include "header.h"** in the context of the C preprocessor?
4. What does the **#define** directive do in the C preprocessor, and how is it used to create macros?
5. Explain the use of the **#ifdef**, **#ifndef**, **#else**, and **#endif** directives in conditional compilation with the C preprocessor.
6. What is the purpose of the **#undef** directive in the C preprocessor, and when is it used?
7. How can you concatenate two preprocessor macros to create a new macro in C?
8. Describe the purpose and usage of the **#ifdef**, **#ifndef**, **#elif**, and **#endif** directives in conditional compilation.
9. How does the **#error** directive work in the C preprocessor, and when might it be useful?
10. What is the **#pragma** directive in the C preprocessor, and what are its common applications?
11. Explain the concept of macro substitution and macro expansion in the C preprocessor.
12. How can you define and use function-like macros in C, and what are their advantages?

13. Discuss the role of the **##** operator in macro expansion and its use in creating concatenated macros.
14. What are include guards, and why are they essential when working with header files and the C preprocessor?
15. How do you conditionally compile code in C using preprocessor directives, and what are the benefits of conditional compilation?
16. What is the purpose of the **#pragma once** directive, and how does it differ from traditional include guards?
17. Explain the potential drawbacks and limitations of using the C preprocessor for code manipulation.
18. How can you pass arguments to macros in C, and what are the rules for using arguments within a macro definition?
19. Describe the differences between object-like macros and function-like macros in C preprocessor usage.
20. What is macro expansion, and how does it occur during the compilation process in C?