C Language Notes

Chapter 0: Introduction to C

What Is C?

C is a procedural programming language. It was initially developed by ***Dennis Ritchie*** in the year **1972**. It was mainly developed as a **system programming language** to write an operating system. The main features of the C language include **low-level memory access**, **a simple set of keywords**, and **a clean style**, these features make C language suitable for system programmings like an operating system or compiler development.

It is a **case sensitive** language. Because it is aware of lower case and upper case. For example “myCar”, “MyCar” and “mycar” are three distinct tokens.

Uses Of C

C language is used to program a vide variety of systems. Some of the uses of C are as follows:

1. Major parts of **Windows**, **Linux** and **other operating systems** are written in C.
2. C is used to **write driver programs** for devices like tablets, printers etc.
3. It is used to **program embedded systems** where programs need to run faster in limited memory(mircowave, cameras etc.)

Chapter 1: Variables, Constants And Keywords

Variables

Variable is the name of memory location which stores the data.

Rules for writing a variable

1. Variables are **case sensitive.**
2. 1st character is **alphabet** or ‘_’.
3. No **comma** or **blank space** is allowed in the name of variable.
4. No other symbol than ‘_’.

Constants

Values that **don’t change(fixed).** There are three types of constants:

1. **Integer Constants:** 1, 2, 787, -98, -9 etc.
2. **Real Constants:** 78.8, 2.4, -9.67, 78.98776 etc.
3. **Character Constants:** ‘a’, ‘h’, ‘#’, ‘$’, ‘S’ etc.

Example

```
int a = 45;
int b = 3, c = 5;

float num = 9.87

char a = "a";
char dollar = "$"

//If we use int for real constant then it will remove the digits after decimal points.

int x = 8.90 // Then it will convert to 8
```

Keywords

Reserved words that have **special meaning to the compiler.**

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

Program Structure

The basic structure of a C program is:

```
#include<stdio.h>

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

Comments

Lines that are not part of a program. Comments are ignored by the compiler when executing the code.

```
// This is a single line comment.
/* This is a multi line comment. */
```

Format Specifiers

1. **Integers - %d |** `printf(”age is %d”, age;`
2. **Real Numbers - %f |** `printf(”value of pi is %f”, pi);`
3. **Characters - %c |** `printf(”star looks like this %c”, ‘*’);`

Take Input From User

“scanf()” function is used to take input from user.

```
int age;
printf("Enter your age : ")
scanf("%d", &a) //This willl take input from the user and assign to age.
```

& - **ampersand** is used to define the address of variable.

Chapter 2: Instructions And Operators

Instructions

There are three types of instructions in program:

1. **Type Declaration Instructions**
2. **Arithmetic Instructions**
3. **Controls Instructions**

Type Declaration Instructions

Declare a variable before using it.

Ex: `int a = 3;`, `int b = c = 3;`, `int x = c + 3;`

Arithmetic Instructions

Used to perform common mathematical operations.

```
a + b // a and b are operands and + is a operators

//Examples
int a = b + c;
int a = b = 5;
int x = y * 4;

int a + b = 3 // This is invalid. There must be single variable on LHS.
```

Note: You cannot use **"^"** to perform exponential operators. To perform exponential operations, you have to use `pow(x,y)` for x to the power y. For using **'pow'** function you have to include **'#include <math.h>'** in you header.

- **Modulator Operator | Modulus - %:** It returns remainder for integers. For example: '**7 % 2**' returns 1.
- **Type Conversion:**

1. Operation b/w int and int gives int.
2. Operation b/w float and int gives float.
3. Operation b/w float and float gives float

- **Operation Precedence:**

All the mathematic operations in C follows operation precedence. It means that it will solve brackets first then it will solve ***, /, %** in an operation and then it will then **+,-** and lastly it will perform =.

- **Associativity(For Same Precedence):** If there are ***, / and %** in the same operation then it will follow left to right associativity.(It will not follow BODMAS)

```
int a = 9 * 7 % 8 * 2 / 2
/*
In this expression, associativity is followed.
1st Step) 9 * 7 = 56
2nd Step) 56 % 8 = 7
3rd Step) 7 * 2 = 14
4th Step) 14 / 2 = 7
So the final value of a is 2.
*/
```

Control Instructions

Used to determine the flow of program:

1. Sequence Control
2. Decision Control
3. Loop Control
4. Case Control

Operators

a. Arithmetic Operators
b. Relational Operators
c. Logical Operators
d. Bitwise Operators
e. Assignment Operators
f. Ternary Operators

Relational Operators

These are relational operators:

1. ==
2. >, <
3. >=, <=
4. !=

Assignment Operators

These are assignment operators

- =, +=, -=, *=, /=, %+

Logical Operators

1. && - **AND**
2. || - **OR**
3. ! - **NOT**

Operation Precedence

Priority	Operator
1	!
2	*, /, *
3	+, -
4	>, <, >=, <=
5	==, !=
6	&&
7	||
8	=

Chapter 3: Conditional Statements

There are two types of conditional statements

1. if-else statements
2. switch statements

If-Else Statements

```
if(condition) {
    //code to be executed when condition is true
} else {
   // code to be executed when condition is not true
}
```

Else is optional. if-else statements can also works without {}.

Condition Operators

Ternary Operator: ***You can also use if-else statement in by line by using ternary operator.***

```
condition ? code to be executed if condition is true : code to be executed if condition is flase;
```

If-Else_If-Else Statements

```
if(condition_1) {
    //code to be executed if condition 1 is true
} else if(condition_2) {
    /*code to be executed if condition 1 is false but
    condition 2 is true*/
} else {
    //code to be executed if both conditions are false.
}
```

Switch Properties

1. Cases can be in any order
2. Nested switch(switch inside switch) is allowed.

Switch Statements

```
switch(number) {

case C1: //do something

break;

case C2 : //do something

break;

default : //do something

}

// C1 and C2 are constants.
```

Chapter 4: Loop Control Statements

Loops are used to repeat some parts of a program. These are of three types:

1. For Loop
2. While Loop
3. Do-While Loop

For Loop

```
for(initialisation; condition; updation) {
    //code
}

//Example
for(int i = 1; i < 10; i++) {
    printf("%d \n", i);
  } // This loop will print numbers from 1 to 9.
```

Important Things

- **Increment Operator:** i++(Increase the value by 1)

++i(Pre Increment) | First increase the value then use it.

i++(Post Increment) | First use the value then increase it.

- **Decrement Operator:** | i--(Decrease the value by 1)

--i(Pre Decrement) | First decrease the value then use it.

i--(Post Decrement) | First use the value then decrease it.

- **Loop counter can be float or even character**

`for(float i = 1.0; i < 5.0; i++) {printf("%f, ", i);}` //This will print → 1.0, 2.0, 3.0, 4.0

`for(char c = 'a'; c < 'z'; i++) {printf("%c, ", c);}` //This will print all alphabets.

- **Infinite Loop**

If we leave condition in for loop blank then the code will execute but the loop will never end, it will keep executing the code until the memory of you cpu will fill and by doing this your pc will crash or if you forget to use i++ or i-- then the loop will never end and keep running.

While Loop

```
while(condition) {
    //code
} //only work if condition is true.

int i = 15;
while(i >= 5) {
    printf("%d \n", i);
    i--;
} //This will print numbers from 15 to 5.
```

Do While Loop

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



/* This code will work like while loop if condition is true
but the code will be executed only once if the condition
if false */
```

Break Statement

Break statement will exit the loop. For example:

```
int i = 0;
do {
    printf("%d", i);
    if(i == 6) {
        break;
    }
    i++;
} while(i < 10); // This loop will print number 0 to 6.
/* This will not print 0 to 10 because when i become 6 the break statement
will break the code */
```

Continue Statement

Continue statement will skip to next iteration.

```
for(int i = 0; i <= 20; i++) {
    if(i % 2 != 0) {
        continue;
    }
    printf("%d \n", i);
} /* This will print all even number from 0 to 20 becuse when i % 2 != 0 it is
a odd number and continue statment skips every odd number*/
```

Nested Statement

Nested Loops mens for loop inside a for loop

```
for (...)
{
    for(...) {

    }
} //First the outer loop is executed then the inner loop will execute.
```

Chapter 5: Functions & Recursion

Functions

Functions is a block of code that performs a particular tasks. It can be used multiple times and it increases the reusability of code.

Syntax of Function

```
#include<stdio.h>

//Function prototype - Tells the compiler that we are declaring a function
void printHello();

int main() {

    //Function call - Tell the compiler to execute the code of the function
    printHello();
    return 0;
}

//Function definition - It contains set of instruction that are executed at the time of function call
void printHello() {
    printf("Hello!");
}
```

Properties of Function

- Execution always starts from **main().**
- A function gets called directly or indirectly from main.
- There can be multiple functions in a program.
- A function can pass only one value at a time.
- Changes to parameters in function don't change the values in calling function because a copy of argument is passed to the function

Types of Functions

There are two types of functions in C language:-

1. **Library Functions -** Special functions inbuilt in C. Ex - `printf()` , `scanf()` .
2. **User-defined Functions -** Functions that are declared and defined by the user.

Passing Values & Arguments to a Function

Functions can take values(parameters) and return some value(return value).

```
//For instance, the function definition of sum can be;
int sum(int a, int b) { //Here a & b are paramenters
    int result;
    result = a + b;
    return result;
}

//Now we can call sum function from call;
int main() {
    sum(2, 3) //Here 2 & 3 are arguments

    return 0;
} // We can get 5 as output
```

Arguments & Parameters

Arguments	Parameters
Arguments are actual values that are passed to the function to make a call.	Parameters are values or variable placeholders in function definition.
The are used to send values.	They are used to receive values.
They are also called actual parameter.	They are also called formal parameters.

Call By Value: *Functions make a copy of values passed to it and then changes it. If we change a value inside a function it will not change inside the main function.*

Recursion

When a functions calls itself, it is called recursion.

Properties of Functions

- Anything that can be done with Iteration, can be done with recursion and vice-versa.
- Recursion can sometimes give the most simple solution.
- Base Case is the condition which stops recursion
- Iteration has infinite loop and recursion has stack overflow.

Chapter 6: Pointers

A **pointer** is a variable that stores the memory address of another variable.

Syntax

- int* ptr
- float *ptr
- char *ptr

Example

```
int i = 5;
int *ptr = &i // * - value at address | & - address of

char a = "*";
char *a = &i;

printf("%p", ptr); // %p is used as a format specifier to pointers
printf("%u", ptr); // but sometimes we use %u beacuse unsinged int gives easier value to understand
```

Pointer to a Pointer

A variable that stores the address of another pointer. It can be declared as:

- int **ptr
- float **ptr
- char **ptr

Example

```
float i = 22.3;
float *ptr = &i;

float *pptr = &ptr;
```

Types of Functions Calls

There are two types of function calls:

1. Call by value
2. Call by reference

Call By Value

Passing value of variables as arguments.

```
int sum(int a, int b);

int main() {
    int x = 3, y = 5;
    sum(x, y);
}
```

In sum function, the values of 3, 4 are passed as arguments means sum function make a copy of x and y(3 and 4) and if we change the value of a and b then nothing happens to variables x and y. This is called **call by value.**

Call By Reference

Passing the address of variables as arguments.

```
int sum(int* a, int* b);

int main() {
    int x = 3, y = 5;
    sum(&x, &y);
}
```

Now since the address of variables are passed to the functions, we can modify the value of a variable in calling function using the ***** and **&** operators. This is called **call by reference.**

Note: The address of arguments passed to a function is different form the actual variable because a copy of argument is passed to it.

```
void function(int a);

int main() {
    int i = 5;
    int* ptr_i = &i;
    void function(i);
}

void function(int a) {
    int* ptr_a = &a;
}

//Here the values of ptr_i & ptr_a are different.
```

Chapter 7: Arrays

An **array** is a collection of similar data types stored at contiguous(continuous) memory locations.

Syntax

- int marks[3];
- float percentage[10];

Input & Output

- **Input -** `scanf("%d", &marks[2];`
- **Output -** `prinf("%d", percentage[]7);`

- char name[7];

Initialization Of Array

```
int marks[] = {89, 43, 98, 56, 78};
int marks[5] = {89, 43, 98, 56, 78};

//The number written inside the closed bracket is the size of array and it is not compulsory to define it.
```

- *The memory reserved by **marks** array is **5x4=20 bytes** because it is a combination of 5 integers and 1 integer takes 4 bytes in memory | float - 4 bytes & char - 4 bytes*

Pointer Arithmetic

A pointer can be **incremented** or **decremented.**

Case 1

```
int age = 45;
int *ptr = &age

/* Let the value of ptr
be 8736882 */

*ptr++

/* Now ptr => 8736882 + 4
= 8736886 */
```

Case 2

```
float price = 87.673;
int *ptr = &price

/* Let the value of ptr
be 8736882 */

*ptr++

/* Now ptr => 8736882 + 4
= 8736886 */
```

Case 3

```
char star = "*";
int *ptr = &star

/* Let the value of ptr
be 8736880 */

*ptr++

/* Now ptr => 8736882 + 1
= 8736883 */
```

Following options can be performed on a pointer-

- Addition of a number to pointer
- Subtraction of a number from pointer
- Subtraction of one pointer from another pointer
- Comparison of two pointer variables

Array is a Pointer

Array is itself a pointer as **arr**(name of array) points toward first elements of that array.

```
int arr[];
int* ptr = &arr[0];
int* ptr = arr; //Both are same
```

Traverse An Array

```
char gender[2] = {"M", "F"}

int* ptr = &gender[1];
printf("%d", *ptr); // This will print F

int length = sizeof(array)/sizeof(array[0]; //You can get length of array by this function
```

Array As Function Argument

```
// function prototype
void printNumbers(array[], int n);
// OR
void printNumbers(*array; int n);

// function call
int arr[9];
printNumbers(arr; 9);
```

Multi-Dimensional Arrays

2D Arrays

- ***Declaration***

```
int arr[][] = { {67, 34}, {43, 102} }
```

- ***Access***

`arr[0][0]` = 67

`arr[0][1]` = 34

`arr[1][0]` = 43

`arr[1][1]` = 102

Chapter 8: Strings

A **string** is a 1D character array terminated by '/0'(Null Character). For example, `char string[] = {'H', 'e', 'l', 'l', 'o', '\0'};`

Initialising Strings

There are two ways to initialise a string in C.

1. `char string[] = {'H', 'e', 'l', 'l', 'o', '\0'};`
2. `char string[] = "Hello";` In this case C adds null character in end automatically.

Strings In Memory

A string is stored just like an array in the memory. Let us take a example, `char name[] = "Akshay";`

Element of string	A	k	s	h	a	y	\0
Address of element	2001	2002	2003	2004	2005	2006	2007

A string is stored in memory just like shown above. Null Character is also stored in memory.

Printing Strings & Taking Input From User

The format specifier for string is **%s**. For instance, `char string[] = "hello";`

Printing String - `printf("%s", string);`

Taking String Output - `scanf("%s", string);` **scanf() is can't input multi-words string with spaces.** So we use some string functions instead of **printf()** and **scanf()**

String Functions

gets(string) - This function will input a string even if it is multi-word but it is dangerous and outdated. So whenever you using this there will be a warning.

puts(string) - This function will print a string.

fgets(string, n, file) - This function is also used to take inputs and it is safe. **n** is the size of string whereas you can write **stdin** in place of file.

Strings Using Pointers

We can also store a string using point like this `char *str "Hello World";` This will store string in the memory & assigned address is stored in the char pointer "str".

If a string is defined like this `char string[] = "Hello World!';` then it can't be reinitialized but if it is defined using pointer like this `char *string = "Hello World!";`

then it can be reinitialized.

Standard Library Functions For Strings

To implement standard library functions for strings, you need to include a header file named **<string.h>** to your program.

1. **strlen(string) -** This will count the length of string excluding '\0'.
2. **strcpy(firstString, secondString) -** This will copy the content of secondString to firstString. firstString should have enough size to store secondString.
3. **strcat(firstString, secondString) -** This function concatenates two string. firstString should be large enough.
4. **strcmp(firstString, secondString) -** This function is used to compare two strings. It will return 0 if two strings are equal. It will return positive value is firstString is greater than second and vice-versa.(ASCII Values) | basically it returns **firstString minus secondString**

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

int main() {
    char str_1[15] = "Akshay";
    char str_2[15] = "Akshit";

    int length = strlen(str_1); // Value of length variable will be 6

    strcpy(str_1, str_2);
    printf("%s", str_1); // This will print the value of str_2 beacuse we have used strcpy() function

    strcat(str_1, str2);
    printf("%s", str_1); // This will print "AkshayAkshit" beacuse strcat() function concatenates two strings

    int compare = strcmp(str_1, str_2);
    printf("%d", compare);
    /*
    This will print negative value beacuse str_2 is greater than str_1(ASCII Values)
    Let us compare str_1 and str_2
    First Letter -> a = a
    Second Letter -> k = k
    Third Letter ->  s = s
    Fourth Letter -> h = h
    Fifth Letter -> a ≠ i
    and subtracting a(97) from h(104) we get -8, so it will return -8
    */

    return 0;
}
```

Chapter 9: Structures

Structures are collection of different data types. Syntax for structures is given below

```
// Creating a structure
struct student {
    int roll_no;
    char name[50];
    int adm_no;
    float cgpa;
}

// Creating a structure valriable
struct student s1;
st.roll_no = 23412;
strcpy(s1.name, "akshay") // we have to copy string as name[50] can't be reinitialized
s1.cgpa = 9.5;
s1.adm_no = 6876;
```

Benefits Of Using Structures

- Save us from creating too many variables.
- Good data management/organization.

Structures in Memory

Structures are stored in contiguous memory locations.

Roll No	Name	**Adm** No	**CGPA**

2010	2014	2114	2118

Arrays Of Structures

```
struct students CSE[100]
struct students ECE[100]

// Access
CSE[0].roll_no = 1562;
CSE[0].cgpa = 8.9;
```

Initialising Structures

Instead of creating a structure variable and declare it in many line we can initialise structures in only one line like this:

```
struct students s1 = {23412, "akshay" 9.5, 6876};
struct students s2 = {23576 "rajat", 8.9, 9867};
struct students s3 = {0} // all the elements are set to zero(null).
```

Pointers to Structures

A pointer to structure can be created as follows:

```
struct student s1;
struct student *ptr;

ptr = &s1

// Now we can print elements as follows;
printf("Roll No: %d", (*ptr).roll_no)
```

Arrow Operator

Instead of writing *(ptr).code, we can use an arrow operator to access structure properties as `ptr->roll_no`

Passing Structures to Functions

We can pass structures to functions like this - `void printData(struct students s1);`

Typedef Keyword

We can use the ***typedef*** keyword to create an alias name for data types in c. ***typedef*** is more commonly used with structures.

```
typedef struct computerScienceEngineering {
    int roll_no;
    char name[50];
    int adm_no;
    float cgpa;
} cse;

// Now we can use the structues as follows
cse student s1 = {12345, "xyz", 5665, 7.6};
```

Chapter 10: File I/O

- **FILE -** container in a storage device to store data

- RAM is volatile

- Contents are lost when program terminates

- Files are used to persist data

Types of files

Text Files	Binary Files
These files contains data in the form of text	These file contains data in the form of binary format(0,1)
Example - .c, .txt, .js etc	Example - .mp3, .exe, .mp4 etc

FILE Pointer

We can perform various options on file like create, open, close, read & write on file.

- **FILE** is a (hidden)structure that needs to be created for opening a file. A FILE ptr that points to this structure & is used to access the file.
- `FILE *fptr;`

```
FILE *fptr;

fptr = fopen("filename", mode); // this is used to open a file

fclose(fptr); // this is used to close a file after opening it
```

File Opening Modes

1. “r” - open to read
2. “rb” - open to read in binary
3. “w” - open to write
4. “wb” - open to write in binary
5. “a” - open to append

Note: *If you open a file in read mode and it doesn't exists then it returns* ***NULL*** *and if you open and file in write mode and it doesn't exists then a* ***new file is created*** *automatically.*

Reading form a file

`fscanf()` is used read data from a file.

```
FILE *fptr;
fptr = fopen("test.txt", read);

int no;
fscanf(fptr, "%d", &no);
/* This will read the data
and store it to no */

fclose(fptr);
```

Writing to a file

`fprintf()` is used to write data to a file.

```
FILE *fptr;
fptr = fopen("test.txt", write);

char str[] = "string";
fprintf(fptr, "%s", str);
/* This will write the
string to file */

fclose(fptr);
```

Note: *If you write on a file then it will erase all the existing data on the file and write the new data on file so if you want to keep existing data on file and write new data then you have to open the file in append mode.*

Reading & Write a Character

`fgetc(file_pointer)` - It is used to read a character from a from a file.

`fputc('char', file_pointer)` - It is used to write a character to a file.

- ***fgetc()*** *will return* ***EOF*** *to show that the file has ended*

Chapter 11: Dynamic Memory Allocation

C is a language with some fixed rules of programming. For example: changing the size of an array is not allowed.

- **Dynamic Memory Allocation(DMA):** It is a way to **allocate memory** to a data structure during **runtime**.

Function for DMA

We need some functions to allocate and free memory dynamically and these functions can be used after including a header file **stdlib.h.**

malloc()

malloc stands for **memory allocation**. It takes number of bytes to be allocated as an input and returns a pointer of type **void**.

```
int *ptr;
ptr = (int *) malloc(5 * sizeof(int));
```

calloc()

calloc stands for **continuous allocation**. It initializes each memory block with a default value of 0.

```
float *ptr;
ptr = (int *) calloc(5, sizeof(int));
```

- Both malloc and calloc return void type pointer so we have to typecast it to type we want using (datatype *) before malloc or calloc.
- If you system has not sufficient memory to allocate, the **NULL pointer** is returned.
- malloc is preferred over calloc because calloc assigns zero to each memory block which leads to usage of more memory.

free()

free function is used to **deallocate the memory** the is allocated using malloc or calloc. **Syntax -** free(ptr).

realloc()

Realloc stand for re-allocation of memory. Sometimes, the allocated memory is insufficient or too much, so we can increase or decrease it using realloc().

```
// Syntax - ptr = realloc(pointer, size)
ptr = realloc(ptr, 3 * sizeof(int));
```