C Programming Language

Contents

CHAPTER 1

INTRODUCTION

- C is a general-purpose programming language created by Dennis Ritchie at the Bell Laboratories in 1972.
- It is a very popular language, despite being old.
- C is strongly associated with UNIX, as it was developed to write the UNIX operating system.

CHAPTER 2

SYNTAX IN C

```
#include <stdio.h>

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

- **Line 1**: `#include <stdio.h>` is a header file library that lets us work with input and output functions, such as printf() (used in line 4). Header files add functionality to C programs.
- **Line 2 :** A blank line. C ignores white space. But we use it to make the code more readable.
- **Line 3 :** Another thing that always appear in a C program, is `main()`. This is called a function. Any code inside its curly brackets `{}` will be executed.
- **Line 4 :** `printf()` is a function used to output/print text to the screen. In our example it will output "Hello World".
- **Line 5 :** `return 0` ends the main() function.
- **Line 6 :** Do not forget to add the closing curly bracket `}` to actually end the main function.

```
#include <stdio.h>

int main() {
  printf("Hello World!\n");
  printf("I am learning C.");
  return 0;
}
```

Output :

```
Hello World!
I am learning C.
```

`\n`: means the newline.

Escape Sequence	Description
"\n"	newline

Escape Sequence	Description
"\t"	Creates a horizontal tab
\\	Inserts a backslash character
\"	Inserts a double quote character

CHAPTER 3

COMMENTS

Comments can be used to explain code, and to make it more readable. It can also be used to prevent execution when testing alternative code.

Comments can be `singled-lined` or `multi-lined`.

Single-line Comments

- Single-line comments start with two forward slashes (//).

```
#include <stdio.h>

int main(){
    //This is a single comment
    printf("Hello World!");
    return 0;
}
```

Multi-line Comments

- Multi-line comments start with `/*` and ends with `*/`.

```
int main(){
    /* The code below will print the words Hello World! to the screen, and it is
amazing.
    */
    printf("Hello World!");
    return 0;
}
```

NOTE : Any text between the comments will be ignored by the compiler.

CHAPTER 4

VARIABLES AND DATATYPES

Variables are containers for storing data values.

Syntax:

```
type variableName = value;
```

Where type is one of C types (such as int), and variableName is the name of the variable (such as x or myName). The equal sign is used to assign a value to the variable.

Declaring a Variable

```
#include <stdio.h>

int main(){
    int a; //Declaration
    return 0;
}
```

Initialising a Variable

```
#include <stdio.h>

int main(){
    int a; //Declaration
    a = 15; // Initialisation
    return 0;
}
```

Declaring and Initialising a Variable in a single Line

```
#include <stdio.h>

int main(){
    int a = 15; // Both are done in a single line
    return 0;
}
```

How to print Vaiable in the Output Screen?

```
#include <stdio.h>

int main(){
    int a = 15;
    printf(a);
    return 0;
}
```

Output : Error

For printing some values we need ***FORMAT SPECIFIERS***.

Format Specifiers

Format specifiers are used together with the `printf()` function to tell the compiler what type of data the variable is storing. It is basically a placeholder for the variable value.

A format specifier starts with a percentage sign `%`, followed by a character.

Foramt Specifiers for some data-types are:

- int = "%d"
- char = "%c"
- double = "%lf"
- long long double = "%lf"
- long long int = "%ld"
- float = "%f"

To debug the upper one is:

```
#include <stdio.h>

int main(){
    int a = 15;
    printf("%d",a);
    return 0;
}
```

Output : 15

How to introduce many variables?

```
#include <stdio.h>

int main(){
    int a,b,c;
    // a = 15; b = 16; c = 17;

    // int a = 1, b = 2, c = 3;

    // a = b = c = 3;
    return 0;
}
```

Rules for naming a Variables :

- Names can contain letters, digits and underscores.
- Names must begin with a letter or an underscore (_).
- Names are case sensitive (myVar and myvar are different variables).
- Names cannot contain whitespaces or special characters like !, #, %, etc.
- Reserved words (such as int) cannot be used as names.

Example:

```
// Create variables
int myNum = 5;             // Integer (whole number)
float myFloatNum = 5.99;   // Floating point number
char myLetter = 'D';       // Character

// Print variables
printf("%d\n", myNum);
printf("%f\n", myFloatNum);
printf("%c\n", myLetter);
```

DATATYPES

The data type specifies the size and type of information the variable will store.

Data Type	Size
int	2 or 4 bytes
char	4 bytes
double	8 bytes
bool	1 byte

CHAPTER 5

Constants

When you don't want others (or yourself) to override existing variable values, use the const keyword (this will declare the variable as "constant", which means unchangeable and read-only).

Example :

```
const int myNum = 15;  // myNum will always be 15
myNum = 10;  // error: assignment of read-only variable 'myNum'
```

The constants are used to declare the values of mathematical constants which will be unchangeable during the code.

Example : Pi, exponential values, modulus values, etc....

NOTE :

- When you declare a constant variable, it must be assigned with a value.

CHAPTER 5

Operators

Operators are used to perform operations on variables and values.

C divides the operators into the following groups:

- Arithmetic operators
- Assignment operators
- Comparison operators
- Logical operators
- Bitwise operators

Arithmetic Operators

Arithmetic operators are used to perform common mathematical operations.
It contains:

- Addition (+)
- Substraction (-)
- Division (/) => It gives the result as the quotient of the division
- Modulo Division (%) => It gives the remainder of the division as the result.
- Increment (++)
- Decrement (--)
- Multiplication (*)

Assignment Operators

Assignment operators are used to assign values to variables.
It contains:

- (=) : equals
- (+=): addition assign
- (-=): substraction assign
- (*=): product assign
- (/=): division assign
- (%=): modulo assign
- (&=): Logical AND assign
- (|=): Logical OR assign
- (^=): Logical XOR assign
- (<<=): Left Shift assign
- (>>=): Right Shift assign

Comparison operators

Comparison operators are used to compare two values.

Note: The return value of a comparison is either true (1) or false (0).

It contains:

- == : Equal to
- != : Not equal to
- < : less than
- (>) : greater than
- (>=) : greater than equal to
- <= : less than equal to

Logical operators

Logical operators are used to determine the logic between variables or values:

It contains:

- && [Logical And]
- || [Logical OR]
- ! [Logical Not]

Bitwise operators

It contains:

- (&) : AND
- (^) : XOR
- (|) : OR

The memory size (in bytes) of a data type or a variable can be found with the `sizeof` operator.

```
int myInt;
float myFloat;
double myDouble;
char myChar;

printf("%lu\n", sizeof(myInt));
printf("%lu\n", sizeof(myFloat));
printf("%lu\n", sizeof(myDouble));
printf("%lu\n", sizeof(myChar));
```

NOTE:

- "%lu" is the format specifier of Unsigned int or unsigned long.

CHAPTER 7

IF-ELSE Statements

C has the following conditional statements:

- Use `if` to specify a block of code to be executed, if a specified condition is true
- Use `else` to specify a block of code to be executed, if the same condition is false
- Use `else if` to specify a new condition to test, if the first condition is false
- Use `switch` to specify many alternative blocks of code to be executed

Syntaxes

if-else statements :

```
if(conditions){
    //code
}
else{
    //code
}
```

if-if-else statements :

```
if(conditions){
    //code
}
if(conditions){
    //code
}
else{
    //code
}
```

if-else if-else statements :

```
if(conditions){
    //code
}
else if(conditions){
    // if statements fails then this will execute
    //code
}
else{
    // if all conditions failed
    //code
}
```

nested if-else (Part-1):

```
if(conditions){
    if(conditions){
        //code
    }
    else{
        //code
    }
}

else{
    if(conditions){
        //code
    }
    else{
        //code
    }
}
```

nested if-else (Part-2):

```
if(conditions){
    if(conditions){
        //code
    }
    else{
        //code
    }
}

else{
    // code
}
```

How to return in if-else statements :

```
if(conditions){
    return (value);
}
else{
    //if the above condition failed then
    return (other_value);
}
```

Short Hand if-else OR Ternary Operator

```
variable = (condition) ? expressionTrue : expressionFalse;
```

Example :

```
int time = 20;
(time < 18) ? printf("Good day.") : printf("Good evening.");
```

CHAPTER 8

SWITCH Statements

Instead of writing many if..else statements, you can use the switch statement.

```
switch(expression) {
  case x:
    // code block
    break;
  case y:
    // code block
    break;
  default:
    // code block
}
```

Working Principle :

- The switch expression is evaluated once
- The value of the expression is compared with the values of each case
- If there is a match, the associated block of code is executed
- The break statement breaks out of the switch block and stops the execution
- The default statement is optional, and specifies some code to run if there is no case match

break Keyword

- When C reaches a break keyword, it breaks out of the switch block.
- This will stop the execution of more code and case testing inside the block.
- When a match is found, and the job is done, it's time for a break. There is no need for more testing.

CHAPTER 9

WHILE LOOP

What is LOOP ?

Loops can execute a block of code as long as a specified condition is reached.

Loops are handy because they save time, reduce errors, and they make code more readable.

Now what is then while Loop ?

The `while` loop loops through a block of code as long as a specified condition is `true`.

Syntax:

```
while (condition) {
  // code block to be executed
}
```

Example :

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

int main(){
    int i = 1;
    while(count < 8){
        printf("COUNT: %d\n",count);
        count++;
    }
    return 0;
}
```

Output :

```
COUNT: 1
COUNT: 2
COUNT: 3
COUNT: 4
COUNT: 5
COUNT: 6
COUNT: 7
```

Do/While Loop

The `do/while` loop is a variant of the `while` loop. This loop will execute the code block once, before checking if the condition is true, then it will repeat the loop as long as the condition is true.

Syntax:

```
do {
  // code block to be executed
}
while (condition);
```

Example :

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

int main(){
    int i = 1;
    do{
        printf("COUNT: %d\n",count);
        count++;
    }while(count < 8)
    return 0;
}
```

Output :

```
COUNT: 1
COUNT: 2
COUNT: 3
COUNT: 4
COUNT: 5
COUNT: 6
COUNT: 7
```

CHAPTER 10

FOR LOOP

When you know exactly how many times you want to loop through a block of code, use the `for` loop instead of a `while` loop.

Syntax:

```
for(initialise; condition; incremental/decremental){
    //code
}
```

Example :

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

int main(){
    int i;
    for(i = 1; i<8; i++){
        printf("COUNT: %d\n", i);
    }
    return 0;
}
```

Output :

```
COUNT: 1
COUNT: 2
COUNT: 3
COUNT: 4
COUNT: 5
COUNT: 6
COUNT: 7
```

CHAPTER 11

FUNCTIONS

- A function is a block of code which only runs when it is called.
- You can pass data, known as parameters, into a function.
- Functions are used to perform certain actions, and they are important for reusing code: Define the code once, and use it many times.

Predefined Functions OR In-built Functions :

- So it turns out you already know what a function is.

Example: main() is a function, which is used to execute code, and printf() is a function; used to output/print text to the screen.

User defined Functions :

- The function which is made by the user and then it is been used in the `main` function is called the user-defined function.
- It is been called in the main function by two different process which is called as `Call by Value` and other is `Call by Reference`.

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

void sum_1(int a, int b){
    int c = a+b;
    printf("VOID WALA: ");
    printf("%d", c);
}

int sum_2(int a,int b){
    int c = a+b;
    printf("INT WALA: ");
    return (c);
}

int main(){
    int a = 2;
    int b = 2;
    sum_1(a,b);
    printf("\n");
    int ans = sum_2(a,b);
    printf("%d",ans);
    return 0;
}
```

NOTE : Here sum_1 and sum_2 are the user defined functions which is been called in the main function.

Call a Function :

- Declared functions are not executed immediately. They are "saved for later use", and will be executed when they are called.
- To call a function, write the function's name followed by two parentheses `()` and a semicolon `;`.

Code Example :

```
// Create a function
void myFunction() {
  printf("I just got executed!");
}

int main() {
  myFunction(); // call the function
  return 0;
}

// Outputs "I just got executed!"
```

What is inside the function parenthesis() ?

- In the function parenthesis it contains the parameters for calculating further issues.

Parameters and Arguments

- Information can be passed to functions as a parameter. Parameters act as variables inside the function.
- Parameters are specified after the function name, inside the parentheses. You can add as many parameters as you want, just separate them with a comma.

Syntax :

```
returnType functionName(parameter1, parameter2, parameter3) {
  // code to be executed
}
```

Example Code :

```
void myFunction(char name[]) {
  printf("Hello %s\n", name);
}

int main() {
  myFunction("Liam");
  myFunction("Jenny");
  myFunction("Anja");
  return 0;
}

// Hello Liam
// Hello Jenny
// Hello Anja
```

Multiple Parameters

- Inside the function, you can add as many parameters as you want.

Example Code :

```
void myFunction(char name[], int age) {
  printf("Hello %s. You are %d years old.\n", name, age);
}

int main() {
  myFunction("Liam", 3);
  myFunction("Jenny", 14);
  myFunction("Anja", 30);
  return 0;
}

// Hello Liam. You are 3 years old.
// Hello Jenny. You are 14 years old.
// Hello Anja. You are 30 years old.
```

Return Values

The `void` keyword, used in the previous examples, indicates that the function should not return a value. If you want the function to return a value, you can use a data type (such as int or float, etc.) instead of void, and use the return keyword inside the function.

Example Code :

```
int myFunction(int x) {
  return 5 + x;
}
```

```
int main() {
  printf("Result is: %d", myFunction(3));

  return 0;
}

// Outputs 8 (5 + 3)
```

Function Declaration and Definition

A function consist of two parts:

- **Declaration:** the function's name, return type, and parameters (if any).
- **Definition:** the body of the function (code to be executed).

Syntax :

```
void myFunction() { // declaration
  // the body of the function (definition)
}
```

Example_1 :

```
// Function declaration
void myFunction();

// The main method
int main() {
  myFunction();  // call the function
  return 0;
}

// Function definition
void myFunction() {
  printf("I just got executed!");
}
```

Example_2 :

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

int main() {
  int result = myFunction(5, 3);
  printf("Result is = %d", result);
  return 0;
}
```

CHAPTER 12

RECURSION

- Recursion is the technique of making a function call itself. This technique provides a way to break complicated problems down into simple problems which are easier to solve.

Example :

```
int sum(int k);

int main() {
  int result = sum(10);
  printf("%d", result);
  return 0;
}

int sum(int k) {
  if (k > 0) {
    return k + sum(k - 1);
  } else {
    return 0;
  }
}
```

Example Explained :

- When the `sum()` function is called, it adds parameter k to the sum of all numbers smaller than k and returns the result.
- When k becomes 0, the function just returns 0. When running, the program follows these steps:

```
10 + sum(9)
10 + ( 9 + sum(8) )
10 + ( 9 + ( 8 + sum(7) ) )
...
10 + 9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 + sum(0)
10 + 9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 + 0
```

- Since the function does not call itself when k is 0, the program stops there and returns the result.

CHAPTER 13

Arrays

- Arrays are used to store multiple values in a single variable, instead of declaring separate variables for each value.
- To create an array, define the data type (like int) and specify the name of the array followed by square brackets [].
- To insert values to it, use a comma-separated list, inside curly braces.

```
int myNumbers[] = {25, 50, 75, 100};
```

Access the Elements of an Array

- To access an array element, refer to its index number.
- Array indexes start with 0: [0] is the first element. [1] is the second element, etc.
- This statement accesses the value of the first element [0] in `myNumbers`.

```
int myNumbers[] = {25, 50, 75, 100};
printf("%d", myNumbers[0]);

// Outputs 25
```

Change an Array Element

- To change the value of a specific element, refer to the index number:

```
myNumbers[0] = 33;
```

Loop Through an Array

- You can loop through the array elements with the for loop.

```
int myNumbers[] = {25, 50, 75, 100};
int i;

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

Example During Class

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


int main(){
    int n; // declare of variable
    int rishav[100] = {1,2,3,4,5,6,7}; //Intialise during declaring

    // Initialise by user
    int tiwari[100];

    int i;

    //user input to array
    printf("Enter the values in array: \n");
    for(i=0; i<10; i++){
        scanf("%d", &tiwari[i]);
    }

    printf("Printing Array: \n");

    // printing the array values
    for(i=0; i<10; i++){
        printf("%d\n", tiwari[i]);
    }

    /*
    printf("%d\n", rishav[0]);
    printf("%d\n", rishav[1]);
    printf("%d\n", rishav[2]);
    printf("%d\n", rishav[3]);
    printf("%d\n", rishav[4]);
    printf("%d\n", rishav[5]);
    printf("%d\n", rishav[6]);
    */
    return 0;
}
```

CHAPTER 14

MultiDimensional Arrays

- The two-dimensional array can be defined as an array of arrays. The 2D array is organized as matrices which can be represented as the collection of rows and columns. However, 2D arrays are created to implement a relational database lookalike data structure. It provides ease of holding the bulk of data at once which can be passed to any number of functions wherever required.

Declaration of 2 dimensional Array

- The syntax to declare the 2D array is given below.

```
data_type array_name[rows][columns];
```

Example :

```
int twodimen[4][3];
```

- Here, 4 is the number of rows, and 3 is the number of columns.

Example During Class

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

int main(){
    int n,row,column;
    printf("Enter the number of the elements: \n");
    scanf("%d",&n);
    int rishav[n][n];
    printf("Enter the elements into the array: \n");
    for(row = 0; row<n; row++){
        for(column = 0; column<n; column++){
            scanf("%d", &rishav[row][column]);
        }
    }
    printf("Printing Array: \n");
    for(row = 0; row<n; row++){
        for(column = 0; column<n; column++){
            printf("%d\n", rishav[row][column]);
        }
    }
    return 0;
}
```

CHAPTER 15

Strings

- Strings are used for storing text/characters.
- For example, "Hello World" is a string of characters.

```
char greetings[] = "Hello World!";
```

- To output the string, you can use the `printf()` function together with the format specifier `%s` to tell C that we are now working with strings.

```
char greetings[] = "Hello World!";
printf("%s", greetings);
```

Access Strings

- Since strings are actually arrays in C, you can access a string by referring to its index number inside square brackets [].
- This example prints the first character (0) in greetings:

Example :

```
char greetings[] = "Hello World!";
printf("%c", greetings[0]);
```

Note : We have to use the %c format specifier to print a single character.

Modify Strings

- To change the value of a specific character in a string, refer to the index number, and use single quotes.

Example :

```
char greetings[] = "Hello World!";
greetings[0] = 'J';
printf("%s", greetings);
// Outputs Jello World! instead of Hello World!
```

Another Way Of Creating Strings

- In the examples above, we used a "string literal" to create a string variable. This is the easiest way to create a string in C.

Example :

```
char greetings[] = {'H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd', '!',
'\0'};
printf("%s", greetings);
```

Differences

- The difference between the two ways of creating strings, is that the first method is easier to write, and you do not have to include the \0 character, as C will do it for you.
- You should note that the size of both arrays is the same: They both have 13 characters (space also counts as a character by the way), including the \0 character.

Example :

```
char greetings[] = {'H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd', '!',
'\0'};
char greetings2[] = "Hello World!";

printf("%lu\n", sizeof(greetings));   // Outputs 13
printf("%lu\n", sizeof(greetings2));  // Outputs 13
```

Example During Class

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

int main(){
    char c[20] = "Rishav Pandey";
    printf("%s\n", c);

    char name[20];
    printf("Enter your name: ");
//  scanf("%s", &name); //pura naame ya string print nhi krega. Space ke baad
terminate kr jayega.
    fgets(name, sizeof(name),stdin); // pura naame ya string print kr dega
//  printf("%s\n", name);
    puts(name);
    return 0;
}
```

CHAPTER 16

Memory Address

- When a variable is created in C, a memory address is assigned to the variable.
- The memory address is the location of where the variable is stored on the computer.
- When we assign a value to the variable, it is stored in this memory address.
- To access it, use the reference operator (&), and the result will represent where the variable is stored.

Example :

```
int myAge = 43;
printf("%p", &myAge); // Outputs 0x7ffe5367e044
```

Note : The memory address is in hexadecimal form (0x..). You probably won't get the same result in your program.

You should also note that &myAge is often called a "pointer". A pointer basically stores the memory address of a variable as its value. To print pointer values, we use the %p format specifier.

Why is it useful to know the memory address?

- Pointers are important in C, because they give you the ability to manipulate the data in the computer's memory - this can reduce the code and improve the performance.

NOTE : Java doesn't have pointers.

Example During Class

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

int main(){
    int i;
    int a;
    printf("Address of i: %x\n", &i);
    printf("Address of a: %x\n", &a);
    return 0;
}
```

Output:

```
Address of i: 62fe1c
Address of a: 62fe18
```

CHAPTER 17

Pointers

- Pointers (pointer variables) are special variables that are used to store addresses rather than values.

Creating Pointers

- You learned from the previous chapter, that we can get the memory address of a variable with the reference operator &.

Example:

```
int myAge = 43; // an int variable

printf("%d", myAge);  // Outputs the value of myAge (43)
printf("%p", &myAge); // Outputs the memory address of myAge (0x7ffe5367e044)
```

- In the example above, &myAge is also known as a pointer.
- A pointer is a variable that stores the memory address of another variable as its value.
- A pointer variable points to a data type (like int) of the same type, and is created with the operator. The address of the variable you're working with is assigned to the pointer.

```
Example
int myAge = 43;     // An int variable
int* ptr = &myAge;  // A pointer variable, with the name ptr, that stores the
address of myAge

// Output the value of myAge (43)
printf("%d\n", myAge);

// Output the memory address of myAge (0x7ffe5367e044)
printf("%p\n", &myAge);

// Output the memory address of myAge with the pointer (0x7ffe5367e044)
printf("%p\n", ptr);
```

Example explained :

- Create a pointer variable with the name ptr, that points to an int variable (myAge). Note that the type of the pointer has to match the type of the variable you're working with.
- Use the & operator to store the memory address of the myAge variable, and assign it to the pointer.
- Now, ptr holds the value of myAge's memory address.

Dereference

- In the example above, we used the pointer variable to get the memory address of a variable (used together with the & reference operator).
- However, you can also get the value of the variable the pointer points to, by using the * operator (the dereference operator).

```
Example
int myAge = 43;     // Variable declaration
int* ptr = &myAge;  // Pointer declaration

// Reference: Output the memory address of myAge with the pointer (0x7ffe5367e044)
printf("%p\n", ptr);

// Dereference: Output the value of myAge with the pointer (43)
printf("%d\n", *ptr);
```

Note : The `*` sign can be confusing here, as it does two different things in our code:

- When used in declaration (int* ptr), it creates a pointer variable.
- When not used in declaration, it act as a dereference operator.

Example During Class

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

int main(){
    int j = 40;
    int *p;
    p = &j;
    printf("Address of j: %x\n", &j);
    printf("Address of p: %x\n", p);
    return 0;
}
```

Output:

```
Address of j: 62fe14
Address of p: 62fe14
```