PART - 1

# C PROGRAMMING HANDBOOK

Basics And Workflow In A Succinct Manner

**BY DEBMALYA SUR**

DEBMALYA SUR

# C Programming Handbook

*Basics And Workflow In A Succinct Manner*

*First edition*

*This book is dedicated to my parents. Their never-ending sacrifices and love always motivate me for working hard. One day I will surely reach my dream.*

# Contents

# Preface

Hello readers, glad to see you all over here. So as you can see, it's a handbook on C, so here I will going to discuss or recap the important points regarding the C programming language.

## My aim for this book

> *I am aiming to deliver my knowledge in this field in a compact and succinct manner in this book. If you follow this book you will find the most important point regarding the C programming language in one place, so I will focus more on the application part and will touch on the internal activities part where there will be a need and I will skip some part(s) (like installation part) which is freely available on the blogs or YouTube, so I am going to focus on something on which tutors don't like to focus or just skips those parts.*

## Focus on this part

> *This is going to be a 3 series book, where this is the first part of the book. In this part, I will be focused on the basics and the internals of the control flow. After reading this book, I will suggest you read the rest of the 2 parts*

*of this series, thus you can acquire a good grasp of the C programming language.*

## What is special in this book?

*Well, here you can find a lot of special things, but you can find two extraordinary things inside this book*

✓ *The book is written in too much user-friendly manner and designed in such a manner thus it can help beginners as well as professionals.*

✓ *You can find in-depth concepts of some important topics. As the name of the book suggests that it's a handbook, thus in some cases I will not go too in-depth but on those spots, I will provide external resources that can surely help you.*

✓ *At the end this series, most probably in the last part, you will find how to interact with harwares and the advanced concepts related to the C programming language.*

## Who will get benefited from this book?

*Anyone who is interested to learn the C programming language can use this book and this book will be helpful to them if you have a basic knowledge about the C programming language or you are an expert in this field, then you can refer to this book for a quick revision or just*

*go through all important points in one glance*

> *So without wasting any more time, let's begin our journey...*

# I

# Prerequisite

*I am creating this part to give a brief introduction along with other needy things that you have to know before driving deep into C. In this part of the book, You will see 6 chapters. Without wasting further time, let's start Part I of this book. So, see you in chapter 1, i.e. A Brief Introduction.*

# 1

# A Brief Introduction

Let's begin a new journey in the C programming language, here I am skipping the installation part as we discussed before. So let's write our first C program

```
# include <stdio.h>
int main() {
  printf("Hello World");
  return 0;
}
```

Here we go, as an output of the above program, you will see *Hello World* get printed on your screen.

Okay, now you might be wondering how actually things are going on? We are writing a program which is similar to our normal English language and we know that computers only understand only two bits those are 0's and 1's... then how it is really happening!

If these questions are arising in your mind, then believe me things are going to be complicated but if once you crack the actual mechanism which is going behind the scene, then you will be into the new world of computers where every single thing is happening with the help of just 2 bits 0's and 1's. So in my next chapter, I will try to discuss the behind the scene mechanism, so if you guys are really interested then I will highly recommend you to read the next chapter... otherwise, you can jump directly to Chapter 3.

# 2

# Behind the Scene

Before moving forward, I will like to say that understanding the internal mechanism isn't easy at all for a beginner, so if you are a beginner, don't be upset, I will recommend you to go through this chapter at least once, no matter how much you have consumed but studies say that human brains are more attentive towards images, thus if you now see the images and the flow charts mentioned below, it will be stored in your memory no matter you understand the concept or not... so in the future when you will learn the core concepts of operating system and compiler design, you will surely find the links between those concepts and these flow charts or images.

Now, let's drive into the internally mechanism. Let's assume we have our first C program, for simplicity let me rewrite that here also.

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

```
  printf("Hello World");
  return 0;
}
```

Now, let's assume that we have stored this program named as main.c

Now, if you compile and run this file, the internal flow will be like this:

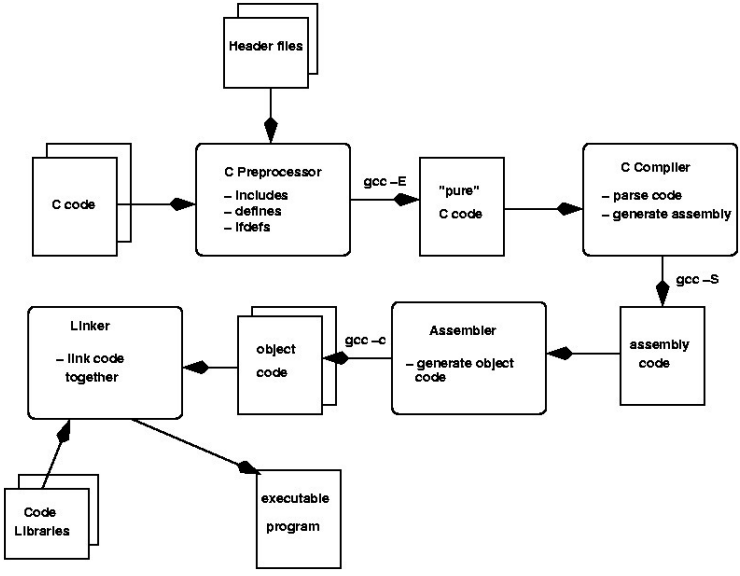



*Image 1: Compilation and making of the executable file of main.c* [1]

Let me discuss each stage briefly,

## Preprocessing Directive

First, you can see # *include <stdio.h>* at the very beginning of the code. So, it's called the preprocessing directive. In a nutshell, here will be the declaration of the functions like *printf* which we have used in this program, so if we do not use # *include <stdio.h>* then we can't able to use the function *printf*, so in the preprocessing stage the whole code of the preprocessing directive is being replaced with the declaration # *include <stdio.h>*

## Compilation Stage

Then this code will be given to the C compiler, there are mainly 6 stages inside the compiler,

1. lexical analysis phase
2. syntax analysis phase
3. semantic analysis phase
4. the intermediate code generation phase
5. code optimization phase
6. target code generation phase

and along with these 6 stages compiler uses two more data structures to store the auxiliary information, the symbol table and the error handling table. As an output of this stage, we will get assembly-level code and this assembly-level code will be dependent on the machine architecture and how the underlined hardware is organized.

## Assembler

Here we will have an input of assembly-level code that is machine-dependent and as an output, we will get a stream of 0's and 1's *i.e.* the set of instructions, which will be given as an input to the next stage named the linking stage.

## Linking Stage

Now, the output of the assembler stage is being linked with the already compiled object code and the starting instruction address of the program will be stored on the program counter (PC), and as an output, we will get an executable file.

## Loading Stage

Now, the whole program will be translated into a process *i.e.* loader will load the instructions onto the RAM or the physical memory and the PC value will be loaded onto the CPU register from where the instruction will be fetched, and now the processor will start executing the process instruction by instruction.

* * *

Now, you might be wondering that till now just our code has started executing on the CPU, but how *Hello World* will be printed

on the screen?

Well, it's interesting and it's possible just because of the operating system. Please see the image below

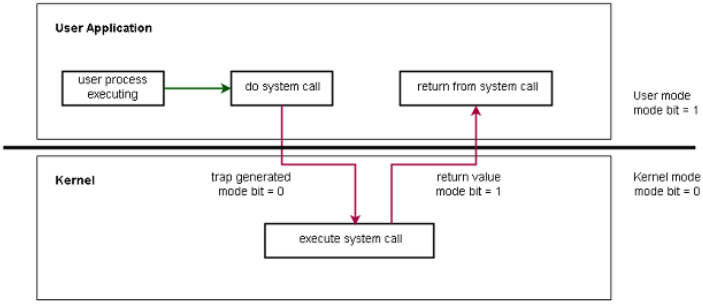



*Image 2: user mode to kernel mode and vice versa* [2]

in our program, *printf* function will internally use a system call which essentially will generate a TRAP through which the mode of the program will be changing from user mode to kernel mode just toggling the bit 1 to 0 of the program status word (PSW) and then OS will execute the system call and will interact with the system hardware and according to the instruction, the OS will show *Hello World* on the peripheral device, here monitor, and after this, the OS will change the mode bit to 1 thus again the process can run in its user mode.

This is how the above code will be executed. I know it's a bit complicated, but at the same time, it's very much important for you to know the flow of control of the execution. If you are a beginner and if you have reached the end of this chapter

but hardly consumed the stuff discussed in this chapter, then don't worry, in this chapter we have discussed the concepts related to the operating system, compiler design, and computer organization & architecture, thus when in the future you will learn these concepts, believe me, everything will be placed in its position.

# 3

# Identifiers & Keywords

We all have names by which our relatives or friends are calling us, similarly in the programming language also programmers have to define a lot of variables, functions, structures, and unions thus to identify them uniquely, programmers have to give them unique names, and these names are called as identifiers.

There are some names that are already defined in the C standards, those are known as keywords. In C, there are 32 such keywords, those are given as follows

| 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 |

*Image 3: Keywords in C* [3]

Now it's time for an example

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

int addition(int x, int y)
{
    int result;
    result = x + y;
    return result;
}

int main()
{
    int number_1 = 10;
    int number_2 = 12;
    double number_3 = 50;

    int add = addition(number_1, number_2);

    printf("Addition Result: %d\n", add);
```

```
    return 0;
}
```

Here are the lists of identifiers of the above program:

1. number_1 (local to the main function)
2. number_2 (local to the main function)
3. number_3 (local to the main function)
4. add (local to the main function)
5. addition
6. x (local to the addition function)
7. y (local to the addition function)
8. result (local to the addition function)

Here are the lists of identifiers of the above program:

1. int
2. double
3. return
4. main

# 4

# Variables & Constants

Variables mean an identifier whose value can be changed. Let's discuss how we can define a variable

## How to define a Variable

```
// Integer Variable
int var1 = 95;

// Double Variable
double var2 = 15.23;

// Character Variable
char var3 = 'A';
```

For our need, we can change the value of the variables inside our program, thus variables play a vital role in our program.

Now, let's discuss the constants. Constants are those identifiers

whose values once declared can't be changed throughout the program. If we forcefully want to change the value of the constant in the program, then it will give a compile-time error.

## How to define a Constant

```
const double PI = 3.14;
PI = 2.9; //It will be Compile Time Error
```

Usually, the mathematical constants are declared as the constants in the program.

# 5

# C Data Types

Till now we are able to declare an identifier and also see how to assign values for those identifiers. Now, let's define another time. So, we are defining a variable that will store an integer of value 5.

```
int var; // line 1
var = 5; // line 2
```

So, in *line 1*, we are declaring a variable named *var* that can store data of an integer. This is satisfied when we add the *int* keyword before the variable name. Here, *int* is the data type of the variable *var*.

In a nutshell, data type determines the type and the size of the data associated with the variable. In this case,

- The type of the data: Integer
- The size of the data: sizeof(int) = 4Bytes

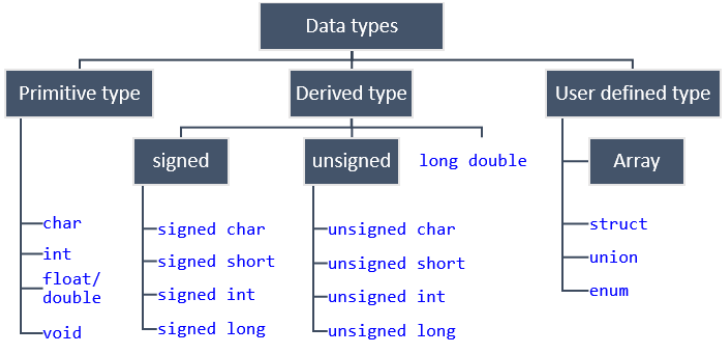



*Image 4: Data Types in C* [4]

At this moment, we are already familiar with some primitive data types, so let's discuss the remaining ones

## Void

Void means nothing, or you can consider it as "no type". Note that, we can't create variables of *void* type, but for the function, the return type may be *void*, which essentially means that the function returns nothing. The void plays an important role in the pointer. I will discuss in the pointer chapter, that void pointers have their own flexibility and importance.

## Signed and Unsigned

In the signed data types, we can represent negative numbers as well as positive numbers but for the unsigned data types, we can only represent non-negative numbers.

Now, in the flow, we will learn the user-defined data types. Before ending this chapter, let me introduce one operator named sizeof()

## Sizeof() Operator

By using this operator we can get the information that how much memory is stored by the variable. Consider the following example

```
#include <stdio.h>
int main() {
  short var1;
  long var2;
  long long var3;
  long double var4;

  printf("size of short = %lu bytes\n", sizeof(var1));
  printf("size of long = %lu bytes\n", sizeof(var2));
  printf("size of long long = %lu bytes\n",
  sizeof(var3));
  printf("size of long double= %lu bytes\n",
  sizeof(var4));

  return 0;
}
```

The output of the above code will be like

```
size of short = 2 bytes
size of long = 8 bytes
size of long long = 8 bytes
size of long double= 16 bytes
```

> *At this point, we have a good understanding of the basics, so in the next chapter I will discuss operators, which are the most important topics in the whole discussion, so I will go into some details there. See you in chapter 6...*

# 6

# Operators in C

So, it's the perfect time to go through the operators in the C programming language. First of all, what is an Operator? well, an operator is a symbol that can operate on a value or a variable. For example, / is the division operator and this operator operates on two numbers (it can be integers or doubles). There are too many operators in C.

Before seeing the whole list we have to know two more terms, those are *precedence order* and *associativity.*

## Precedence Order

In simple terms, it's nothing but the priority of the operators. Let me give you an example

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

```
int main()
{
int a = 2, b = 3, c = 5;
int result = a + b / c;

printf("The Result Is: %d", result);

return 0;
}
```

If I ask you what will be the output of the above code? Then the answer is depending upon one thing *i.e.* whether the above expression is (2 + 3) / 5 or 2 + (3 / 5). Keep thinking, you will get your answer at the end of this chapter.

## Associativity

If there are different operators and they are not given brackets then that problem can be solved by precedence order, but what if the same operators are given in the expression only and no particular brackets are given then how we can solve this problem? Well, let me give an example

```
#include <stdio.h>

int main()
{
int a = 9, b = 3, c = 2;
int result = a / b / c;

printf("The Result Is: %d", result);
```

```
return 0;
}
```

what will be the output? It all depends on the expression evaluation pattern, right? whether the *result* expression will be considered as (9 / 3) / 2 or 9 / (3 / 2)? Keep thinking, the answer to this question also is in the below paragraph.

Hopefully, the problem is clear to you. So, now answer time...

C programming language follows a standard precedence order and corresponding associativity order of the operators, so the compiler has to be strict on that standard. The following table is that standard.

| OPERATOR | TYPE | ASSOCIAVITY |
|---|---|---|
| ( ) [ ] . -> | | left-to-right |
| ++ -- + - ! ~ (*type*) * & sizeof | Unary Operator | right-to-left |
| * / % | Arithmetic Operator | left-to-right |
| + - | Arithmetic Operator | left-to-right |
| << >> | Shift Operator | left-to-right |
| < <= > >= | Relational Operator | left-to-right |
| == != | Relational Operator | left-to-right |
| & | Bitwise AND Operator | left-to-right |
| ^ | Bitwise EX-OR Operator | left-to-right |
| \| | Bitwise OR Operator | left-to-right |
| && | Logical AND Operator | left-to-right |
| \|\| | Logical OR Operator | left-to-right |
| ? : | Ternary Conditional Operator | right-to-left |
| = += -= *= /= %= &= ^= \|= <<= >>= | Assignment Operator | right-to-left |
| , | Comma | left-to-right |

*Image 5: Precedence Order & Associativity of the Operators of C Programming Language* [5]

Now it's the time to come to the solutions to those above questions.

For the *first question*, as the precedence order of operator / is higher than + thus, *2 + (3 / 5)* will be the ultimate expression for the variable *result*. So, the value of the expression will be 2 so, the output of program 1 will be

```
The Result Is: 2
```

For the *second question*, here all three operators are division or / operator, thus here we have to solve the problem using the associativity order of the / operator. See the above table, you will see division operator is left-to-right associative. So, the current expression evaluation order of the variable *result* will be *(9 / 3) / 2*, *i.e.* the value of the expression will be 1 so, the output of program 2 will be

```
The Result Is: 1
```

> *So, here we have reached the end of this chapter as well as the end of this part I. Hopefully, this particular part is clear to you because it plays a great role in the upcoming section of this book. In the next part, we will jump to the Flow Control section, where we will discuss how we control the flow of the execution of the program. I am going to discuss first the if...else statement in chapter 7. So, See you in the next part's first chapter...*

* * *

# II

# The Flow Control

*You are already aware of the basics at this point, so moving further... now we are ready to see how to control the flow of our program.*

*In this part, we will see the if... else statement, loops, switch... case statement, goto statement, break & continue clause. So see you in the first chapter of this part i.e. if... else in C.*

# 7

# if... else in C

Let's begin this chapter with a small example. We all have a simple logic, suppose the weather is rainy, then I will take an umbrella otherwise I will not take the umbrella with me. In other words, whether I will take the umbrella outside or not, totally depends on the weather. On this same logic the if... else statement is built.

## Simple if... else Syntax

The simple syntax of the if... else statement is given below.

```
if (condition) {
    statement
}
else {
    statement
}
```

If the condition is satisfied then the statements in the *if* portion

will be executed, otherwise the statements of the *else* portion will be executed directly. In the condition section, there can be any relational expression that ultimately says whether the relational expression is TRUE (any non-zero integer) or FALSE (zero), if the condition returns a TRUE value, then the control will go into the *if* portion otherwise the control will directly jump to the *else* portion.

### Syntax of if... else Ladder

We can build a ladder by using an additional *else if* clause. A simple example is given below.

```
/* Compares two user-given numbers, whether they are
equal, greater than, or less than. */

#include <stdio.h>
int main() {
    int number1, number2;
    printf("Enter two integers (one after another):
    ");
    scanf("%d %d", &number1, &number2);

    //checking if the two integers are equal.
    if(number1 == number2) {
        printf("Result: %d = %d",number1,number2);
    }

    //checking if number1 is greater than number2.
    else if (number1 > number2) {
        printf("Result: %d > %d", number1, number2);
    }

    //checking if both the above expressions are false
```

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

    return 0;
}
```

Now, let me ask you one question. Consider the following C code.

```
/* Compares two user-given numbers, whether they are
equal, greater than, or less than. */

#include <stdio.h>
int main() {
    int number1, number2;
    printf("Enter two integers (one after another):
    ");
    scanf("%d %d", &number1, &number2);

    //checking if the two integers are equal.
    if(number1 == number2) {
        printf("Result: %d = %d",number1,number2);
    }

    //checking if number1 is greater than number2.
    if (number1 > number2) {
        printf("Result: %d > %d", number1, number2);
    }

    //checking if both the above expressions are false
    if (number1 < number2) {
        printf("Result: %d < %d",number1, number2);
    }

    return 0;
}
```

My questions are:

1. Are the ultimate output of the above two given codes for constant inputs the same or different?
2. Whether the number of comparisons, while we are executing the above code, are same or different for all inputs?

Let you think for a second, and then go to the next paragraph.

It's time to see the answers.

1. Yes, the ultimate outputs for both of those cases will be the same.
2. No, the number of comparisons of both the codes will not be the same. In the case of *if... else if... else* statement if the *if* condition is FALSE then only it will check the condition of the *else if* & if the condition of the *else if* is FALSE then only it will execute the *else* portion statements. But, suppose *else if* condition is TRUE, then here it will not jump to the else, after successfully executing the codes of the *else if* section it will directly jump to execute the code after the *else* section. But for the second code, each of those three *if* conditions will be checked. So on average, the number of comparisons will be higher for the second code segment.

Coming to further extension, Let's connect this point with the asymptotic notation perspective. Suppose we have *n* pairs of user inputs, then for the second code the number of comparisons will be $\theta(n)$. But for the same constraints and for the first code the number of the comparisons will be $O(n)$, *i.e.* in the worst case, there is a possibility that for all of those user inputs, control

needs to end up in the *else* section. But in the average case, the number of comparisons will be less than the O(n).

Hopefully, the above concept is clear to you, if not I will highly recommend you to learn about Asymptotic analysis[6] because it is a topic related to algorithms so here I can't discuss it thoroughly.

Coming back to the if... else statement, we can do the nesting here. So C supports the if... else statement inside another if... else statement.

> *So, we have reached the end of this chapter, in the next chapter, we will see about the loops in the C programming language. See you in chapter 8...*

* * *

# 8

# Loops in C

Welcome to this chapter. In this chapter, we will learn the most used concept in C programming which is the loop. So, here we have 3 types of loops, are

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

If you learn one loop then the remaining loops are based on the same logic. Let's begin with the *for* loop.

## for loop in C

A simple *for* loop example is like follow

```
#include <stdio.h>

int main()
{
```

```
int number;

/*
syntax of for loop in C:
for (initializationStatement; testExpression;
updateStatement)
*/

for(number=10; number>0; number--)
    printf("%d\n", number);

return 0;
}
```

The above C code will print 10 to 1 on the screen. Let's see the flow diagram

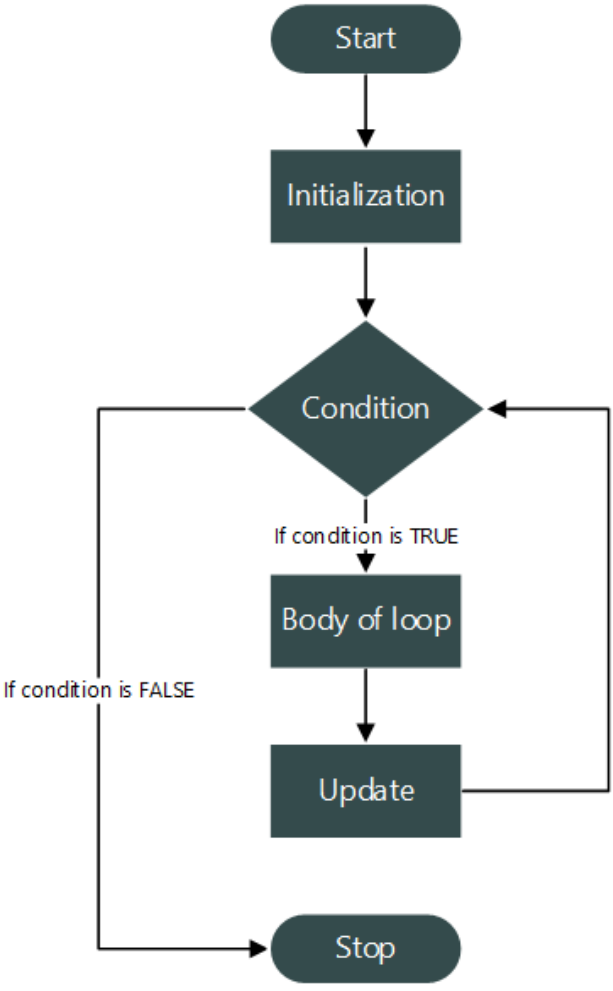



*Image 6: Flow control of for loop in C* [7]

Remember, the initialization is done only once, and then the condition will be checked, if the condition is TRUE then the

control will go into the body of *for* loop and after executing those statements, the control will update the conditional variable, and again will check the condition. The control will go outside of the loop, if and only if the condition returns FALSE.

We will see an example in the next chapter. Now let's move to the *while* loop.

## while loop in C

*While* loop is the same as *for* loop. Almost all programs we have written with *for* loop can be converted to the *while* loop. Let's see its flow diagram

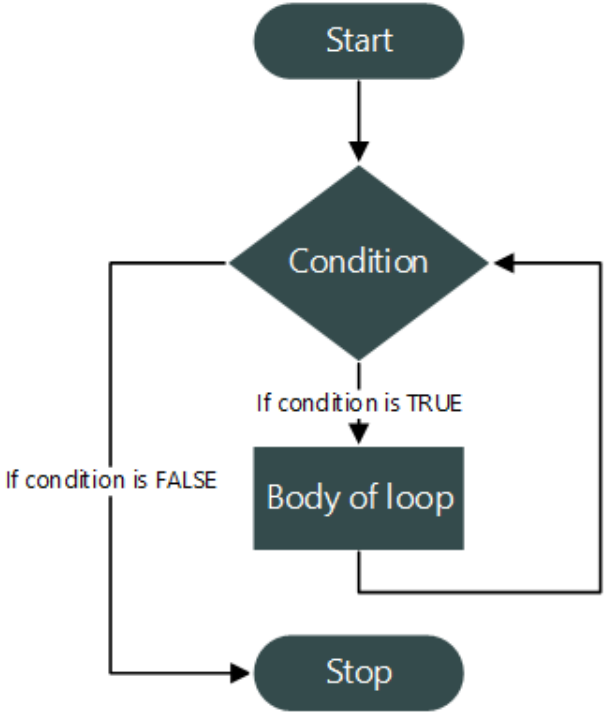



*Image 7: Flow control for while loop in C* [8]

Before going to an example, first, let us quickly discuss the *do... while* loop.

## do... while loop in C

First, let's quickly see the flow chart of *do... while* loop also.

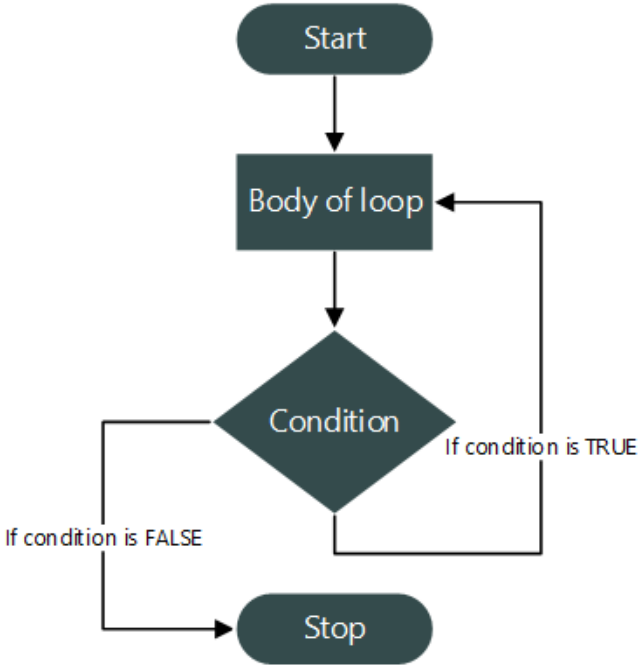



*Image 8: do... while loop in C* [9]

So, the main difference between *while* and *do... while* loop is the *do... while* loop will surely execute once irrespective of whether the condition is TRUE or FALSE but in the case of *while* loop, the number of execution of *while* loop's body is totally dependent upon whether the condition is TRUE or FALSE. In a nutshell, the minimum time the *while* loop's body will be executed is 0 whereas, the minimum time the *do... while* loop's body will be executed will be 1.

> *Here we reached the end of this chapter. In the next chapter, we will see one example & I will show how we can use these three loops interchangeably. So, see you in chapter 9...*

# 9

# Example of Loops

First, let me define a problem statement.

> *You have to print the following pattern on the screen*
> *1 2 3 4 5*
> *1 2 3 4*
> *1 2 3*
> *1 2*
> *1*

It's time to solve the question. First, try it on your own and then see the following approach.

Let's begin. As you can see you have 5 rows and the first row has 5 columns and then the number of columns for a row decreases, thus if we use the loop concept here, then it will be fruitful.

First, let's try the *for* loop. The Below code will be perfect if we use *for* loop.

```
/*
1 2 3 4 5
1 2 3 4
1 2 3
1 2
1
*/

#include <stdio.h>

int main()
{
    int iterations = 5;
    for (int rows = iterations; rows > 0; rows--)
    {
        for (int columns = 1; columns <= rows;
        columns++)
            printf("%d ", columns);
        printf("\n");
    }
return 0;
}
```

Here, the first or outer *for* loop is counting the row number whereas the inner *for* loop is controlling the element that has to be printed on the screen next. So, you can see that the loops can be nested also.

Now, let's try the same problem in only *while* loop form.

```
/*
1 2 3 4 5
1 2 3 4
1 2 3
1 2
```

```
1
*/

#include <stdio.h>

int main()
{
    int iterations = 5;
    int rows = iterations;
    while (rows > 0)
    {
        int columns = 1;
        while (columns <= rows)
        {
            printf("%d ", columns);
            columns++;
        }
        printf("\n");
        rows--;
    }
return 0;
}
```

If you see it carefully, the transformation from *for* loop to *while* loop is quite easy. That's why earlier I have said that *for* loops and *while* loops are interchangeable.

> *Here we have reached to end of this chapter, and the example given in this chapter is clear to you. Remember, loops are the most important concept that is used in C programming while coding. So, in the next chapter, I will discuss break & continue statements. See you in chapter 10...*

# 10

# Break & Continue in C

*Break* and *continue* statements are used in a loop to come out from the loop or to pass that iteration of the loop and jump to the beginning of the next iteration of that loop.

Let me describe this concept using an example,

Let's assume you have to write a function named test, which will take no parameters and it will iterate through 1 to 20 and will print only the even numbers that are below 15.

One way to write the above program using the concept we learned till now is as followed

```
#include <stdio.h>

void test() {
    for (int counter = 1; counter <= 20; counter++) {
        if (counter%2==0 && counter < 15) {
            printf("%d ", counter);
```

```
        }
    }
    printf("\n");
}

// Driver code
int main() {
    test();
    return 0;
}
```

Now first let me write the code of the above problem using *break* and *continue*

```
#include <stdio.h>

void test() {
    for (int counter = 1; counter <= 20; counter++) {
        if (counter < 15) {
            if (counter%2 == 0)
                printf ("%d ", counter);
            else
                continue;
        }
        else
            break;
    }
    printf("\n");
}

// Driver code
int main() {
    test();
    return 0;
}
```

How the above code is working? The continue keyword is used to skip the instructions of the current loop which are below the *continue* statement and jump to the next iteration. On the other hand, the *break* keyword is used to come out from the current loop and execute the next statement that is just outside of that loop. Now let's decode the above code.

The counter is going from 1 to 20 as mentioned in the for loop but essentially the for loop is executing 15 times only!!!

How it's possible right? Let me describe it. The innermost if statement is checking whether the value of the counter is less than 15 or not, thus these many times the loop will be executed because on the value of the counter will be 15 then it will execute else condition and execute *break* statement which essentially moves the control to the outer print statement.

Now suppose the value of the counter is 7, then the condition inside the inner if statement will be false and it will execute the *continue* statement of the inner else statement, which essentially will not execute any further statement of the current loop and directly jump to the increment of the counter variable condition of the *for* loop.

It's all about the *break* and the *continue* statements of the C programming. Just remember, by using the *continue* statement we can skip the instructions below that statement of that loop and can jump to the next iteration, *i.e.* we can skip the remaining instructions of the current loop which are below the *continue* statement, and if we wanna go outside of the current loop then use the *break* statement.

> *So, we have reached the end of this chapter, on the next chapter I will discuss the switch... case, and goto instructions. So, see you in the next chapter...*

# 11

# switch... case & goto in C

Syntax of switch... case statement

```
switch (expression)
{
    case constant1:
      // statements
      // break; /* Optional */

    case constant2:
      // statements
      // break; /* Optional */
    .
    .
    .
    default:
      // default statements
}
```

Flowchart of *switch... case* statement

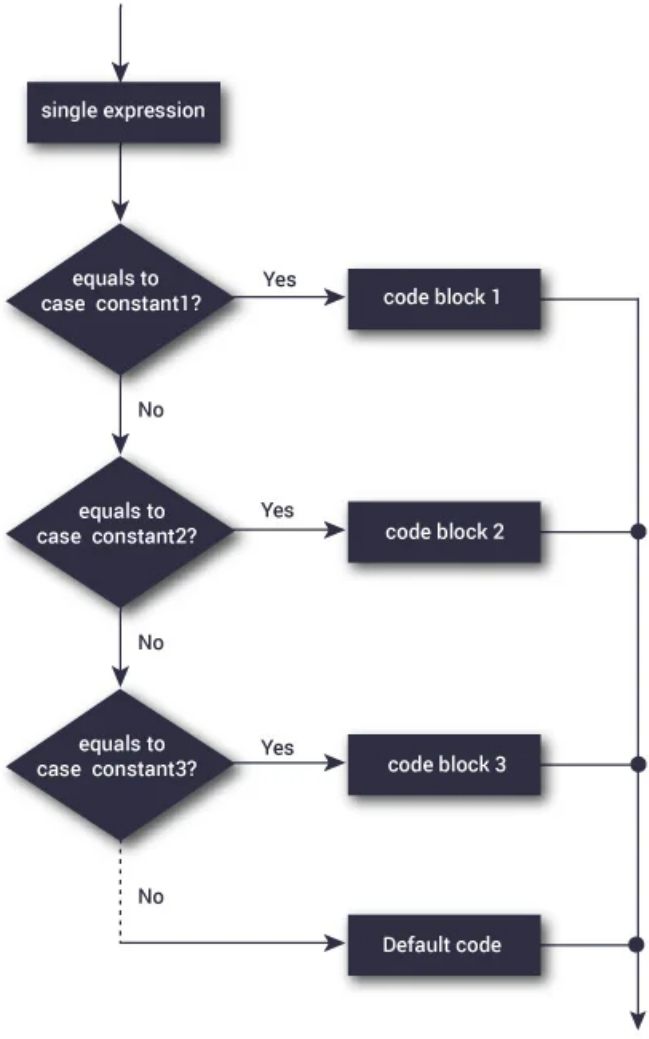



*Image 9: switch... case statement in C*[10]

The *switch... case* statement is used as a supplement to densely nested *if... else* statement. In the expression of the *switch* statement, we will pass a character or integer constant, and the code related to that *case* statement will be executed. We will see an example related to this concept at the end of this chapter.

## Syntax of the goto statement

```
goto label;
... .. ...
... .. ...
label:
statement;
```

If you have studied the microprocessor, then the control flow of the *goto* statement will be exactly the same as the unconditional jump statement. Here, remember one thing, the label which will be mentioned in the *goto* statement, must be defined in the same statement only, otherwise, the C compiler will throw compilation error[11].

The *goto* statement is useful when you want to come out from a deeply nested *if... else statement* or deeply nested *loops*, but in general as it gets converted into an unconditional jump statement in machine-level language, thus it's a better practice not to use the *goto* statement while coding.

## Example of switch... case and goto statement

```
// Implementing Simple Calculator
#include <stdio.h>

void calculator(char op, float num1, float num2) {
    float result = 0;
    switch (op) {
        case '+':
            result = num1 + num2;
            goto end;
        case '-':
            result = num1 - num2;
            goto end;
        case '*':
            result = num1 * num2;
            goto end;
        case '/':
            result = num1 / num2;
            goto end;
        default:
            printf("Please enter a valid operator");
        }
    end:
        printf("%f %c %f = %f", num1, op, num2,
        result);
}

// Driver code
int main() {
    char op;
    printf("Enter an operator between {+, -, *, /} :
    ");
    scanf("%c", &op);

    float num1, num2;
```

```
    printf("Enter two numbers : ");
    scanf("%f %f", &num1, &num2);

    calculator(op, num1, num2);

    return 0;
}
```

The above code is the code of a simple calculator, where if you give 2 numbers and an operator as input then the code will print the output after performing the operation on those 2 numbers.

The key points here are:

1. The condition of the *switch... case* statement should be either an integer constant or a character constant,
2. The label of the *goto* statement should be defined inside that function itself where the *goto* statement is written.

> *And, that's all for this chapter, hopefully, the concept of the switch... case, and the goto statements are clear to you... see you in the announcement section.*

* * *

# III

# Announcement

*You have reached the end of this version of the book.*

*I know it's one-third only. There are topics left like functions, arrays, pointers, strings, and user-defined data types which are covered in the second part of this series, and in the last part, I will discuss file handling, threading, and hardware interaction procedures, etc.*

*Please do visit https://surdebmalya.github.io/ and stay connected with me.*

*Please send your feedback https://surdebmalya.github.io/pages/books/books.html here.*

# Notes

BEHIND THE SCENE

1 The image is taken from https://www.cs.nmsu.edu/~rth/cs/cs271/notes/Compiling.html

2 The image is taken from https://notesformsc.org/operating-system-structure/

IDENTIFIERS & KEYWORDS

3 The image is taken from https://www.educba.com/c-keywords/

C DATA TYPES

4 The image is taken from https://codeforwin.org/2017/08/data-types-in-c-programming.html

OPERATORS IN C

5 The image is taken from: https://www.codingeek.com/tutorials/c-programming/precedence-and-associativity-of-operators-in-c/

IF... ELSE IN C

6 Refer to Wikipedia: https://en.wikipedia.org/wiki/Asymptotic_analysis

LOOPS IN C

7 The image is taken from https://codeforwin.org/2017/09/do-while-loop-c-programming.html

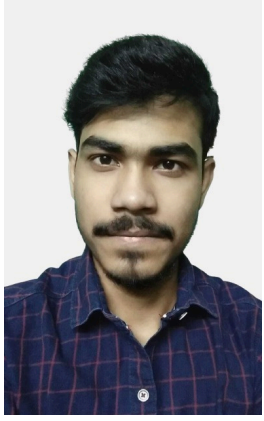
8 The image is taken from https://codeforwin.org/2017/08/while-loop-c-programming.html

9 The image is taken from https://codeforwin.org/2017/09/do-while-loop-c-programming.html

SWITCH... CASE & GOTO IN C

10 The image is taken from https://www.programiz.com/c-programming/c-switch-case-statement

11 Refer to Wikipedia: https://en.wikipedia.org/wiki/Compilation_error



# About the Author

Hello readers, my name is Debmalya Sur, at the time (year-2022) when I am writing this book, I am in my final year of B.Tech in Computer Science & Engineering. I have appeared in GATE Exam 2022 and was able to secure an all India rank of 405, and aiming to do post-graduation in Computer Science from IIT (Indian Institute of Technology)

**You can connect with me on:**

- https://surdebmalya.github.io
- https://github.com/surdebmalya
- https://www.linkedin.com/in/debmalya-sur

*The End*