# C Programming

## Chapter 1 - C Overview

### What is C?

- Since the late 19th century, C has been a popular programming language for general-purpose use.
- C language was developed by Dennis M. Ritchie at bell laboratory in early 1970s
- Its applications are very diverse. It ranges from developing operating systems to databases and all. It is system programming language used to do low level programming (e.g., driver or kernel)
- Even if it's old, it is still a very popular programming language.
- As the whole UNIX operating system was written in C, it has a strong association with the operating system
- C has also been used widely while creating iOS and Android kernels.
- MySQL database is written using C.
- Ruby and Pearl are mostly written using C.
- Most part of Apache and NGINX is written using C.
- Embedded Systems are created using C

### Why should we learn C/ Features of C?

- As mentioned above, it is one of the most popular programming languages in the world.
- Learning any other popular programming language such as Python or C++ becomes a cakewalk already if you know C.
- C is a flexible language and that gets proven by the fact that it can be used in a variety of applications as well as technologies.
- C is very fast when compared to other programming languages be it Java or Python.
- C takes only significant CPU time for interpretation. That is why a lot of Python libraries such as NumPy, pandas, Scikit-learn, etc. are built using C.
- Being close to Machine language, some of its functions include direct access to machine-level hardware APIs.
- It is a structural language (Follows a specific Structure) /Compiled language this point should be added
- It is procedural programming language (POP) Procedural Programming is the use of code in a step-wise procedure to develop applications.

### How is it different from C++?

- The syntax of C++ is almost identical to that of C, as C++ was developed as an extension of C.
- In contrast to C, C++ supports classes and objects, while C does not.
- C gives most of the control to the hand of users. Things like memory allocation and manipulation are totally in the hands of the programmer. Being a flexible language, it provides more access to the programmer because of which it is more efficient.
- C is POP(procedure oriented programming) whereas c++ is OOP(Object oriented programming)

# Chapter 2 - Getting Started with C

## Requirements before you start

To start using C, you need two things:

- A text editor, like Notepad, or an IDE, like VSCode to act as a platform for you to write C code
- A compiler, like GCC to translate the C code you have written which is a high-level language into a low-level language that the computer will understand.

## What is an IDE?

- IDE stands for an Integrated Development Environment.
- It is nothing more than an enhanced version of a text editor that helps you write more efficient and nicer code.
- It helps to differentiate different parts of your codes with different colors and notifies you if you are missing some semicolon or bracket at some place by highlighting that area.
- A lot of IDEs are available, such as DEVC++ or Code Blocks, but we will prefer using VS Code for this course.

## Installing VSCode

- Visit https://code.visualstudio.com/download Click on the download option as per your operating system.
- After the download is completed, open the setup and run it by saving VS Code in the default location without changing any settings.
- You will need to click the next button again and again until the installation process begins.

## What is a Compiler?

- A compiler is used to run the program of a certain language which is generally high-level by converting the code into a language that is low-level that our computer could understand.
- There are a lot of compilers available, but we will proceed with teaching you to use MinGW for this course because it will fulfill all of our requirements, and also it is recommended by Microsoft itself.

## Setting up the compiler

- Visit https://code.visualstudio.com/docs/languages/cpp
- Select C++ from the sidebar.
- Choose “GCC via Mingw-w64 on Windows” from the options shown there.
- Select the install sourceforge option.
- After the downloading gets completed, run the setup and choose all the default options as we did while installing VS Code.

## Setting Path for Compiler

- Go to the C directory. Navigate into the Program Files. Then, open MinGW. Then the bin folder. After reaching the bin, save the path or URL to the bin.

- Then go to the properties of ‘This PC’.
- Select ‘Advance System Settings’.
- Select the ‘Environment Variable‘ option.
- Add the copied path to the Environment Variable.
- And now, you can visit your IDE and run your C programs on it. The configuration part is done.

---

# Chapter 3 - Basic Structure & Syntax

Programming in C involves following a basic structure throughout. Here's what it can be broken down to.

- Pre-processor commands
- Functions
- Variables
- Statements
- Expressions
- Comments

## Pre-processor commands

Pre-processor commands are commands which tell our program that before its execution, it must include the file name mentioned in it because we are using some of the commands or codes from this file.

They add functionalities to a program.

One example could be,

```
#include<math.h>
```

We include math.h to be able to use some special functions like power and absolute. `#include<filename.h>` is how we include them into our programs.

Detailed explanations of everything else in the structure will follow in the later part of the course.

## Header files:

- Collection of predefined/built in functions developed
- It is always declares on heading side of program hence it is called header file
- It is identified with the extension(.h)
- It gets installed while installing IDE(integrated development environment)
- It stores functions as per their categories hence they are called library

## Syntax

An example below shows how a basic C program is written.

```
Declaration of header file
//name of the header files of which functions are been used
main()
/*it is called main function which stores the execution of program*/
{
    //start of the program

    //program statements
```

```
}
//end of the program
```

- Here, the first line is a pre-processor command including a header file stdio.h.
- C ignores empty lines and spaces.
- There is a `main()` function then, which should always be there.

A C program is made up of different tokens combined. These tokens include:

- Keywords
- Identifiers
- Constants
- String Literal
- Symbols

## Keywords

- Keywords are reserved words that can not be used elsewhere in the program for naming a variable or a function. They have a specific function or task and they are solely used for that. Their functionalities are pre-defined.
- One such example of a keyword could be return which is used to build return statements for functions. Other examples are auto, if, default, etc.
- Whenever we write any keyword in IDE their color slightly changes and it looks different from other variables or functions for example in turbo c all keywords are turns into white color.

## Identifiers

- Identifiers are names given to variables or functions to differentiate them from one another. Their definitions are solely based on our choice but there are a few rules that we have to follow while naming identifiers. One such rule says that the name can not contain special symbols such as @, -, *, <, etc.
- C is a case-sensitive language so an identifier containing a capital letter and another one containing a small letter in the same place will be different. For example, the three words: Code, code, and cOde can be used as three different identifiers.

Rules for naming identifier-

1. One should not name any identifier starting with numeric value or symbol. It should start only with underscore or alphabet
2. They should not contain space
3. Giving logical names is recommended as per our program

## Constants

- Constants are very similar to a variable and they can also be of any data type. The only difference between a constant and a variable is that a constant’s value never changes. We will see constants in more detail in the upcoming chapters.

## String literals

- String literals or string constants are a sequence of characters enclosed in double quotation marks. For example, “This is a string literal!” is a string literal. C method `printf()` utilizes the same to format the output.

---

# Chapter 4 - C Comments

Comments can be used to insert any informative piece which a programmer does not wish to be executed. It could be either to explain a piece of code or to make it more readable. In addition, it can be used to prevent the execution of alternative code when the process of debugging is done.

Comments can be singled-lined or multi-lined.

## Single Line Comments

- Single-line comments start with two forward slashes (//).
- Any information after the slashes // lying on the same line would be ignored (will not be executed).

An example of how we use a single-line comment

```c
#include <stdio.h>

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

## Multi-line comments

- A multi-line comment starts with /_ and ends with _/.
- Any information between /_ and _/ will be ignored by the compiler.

An example of how we use a multi-line comment

```c
#include <stdio.h>

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

# Chapter 5 - C Variables

Variables are containers for storing data values In C, there are different types of variables. For example,

- an integer variable defined with the keyword int stores integers (whole numbers), without decimals, such as 91 or -13.
- a floating point variable defined with keyword float stores floating point numbers, with decimals, such as 99.98 or -1.23.
- a character variable defined with the keyword char stores single characters, such as 'A' or 'z'. Char values are bound to be surrounded by single quotes.

## Declaration

We cannot declare a variable without specifying its data type. The data type of a variable depends on what we want to store in the variable and how much space we want it to hold.

The syntax for declaring a variable is simple:

```
data_type  variable_name;
```

OR

```
data_type  variable_name = value;
```

## Naming a Variable

There is no limit to what we can call a variable. Yet there are specific rules we must follow while naming a variable:

- A variable name can only contain alphabets, digits, and underscores(_).
- A variable cannot start with a digit.
- A variable cannot include any white space in its name.
- The name should not be a reserved keyword or any special character.

A variable, as its name is defined, can be altered, or its value can be changed, but the same is not true for its type. If a variable is of integer type, then it will only store an integer value through a program. We cannot assign a character type value to an integer variable. We can not even store a decimal value into an integer variable.

# Chapter 6 - C Data Types & Constants

As explained previously, a variable in C must be a specified data type, and you must use a format specifier inside the printf function to display the value present in the variable

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

## Datatype

It is the type of value which the variable holds.

Here, we will focus on the most basic ones:

| Data Type | Size | Description | Format Specifier |
|---|---|---|---|
| int | 2 or 4 bytes | Stores whole numbers, without decimals | %d or %i |
| float | 4 bytes | Stores fractional numbers, containing one or more decimals. Sufficient for storing 7 decimal digits | %f |
| double | 8 bytes | Stores fractional numbers, containing one or more decimals. Sufficient for storing 15 decimal digits | %lf |
| char | 1 byte | Stores a single character/letter/number, or ASCII values | %c |

Below are some sub datatypes

| Data Type | Size | Description | Format Specifier |
|---|---|---|---|
| short int | 2 bytes | Stores whole numbers, without decimals | %sd |
| long int | 4 bytes | Stores fractional numbers, containing one or more decimals. Sufficient for storing 7 decimal digits | %ld |
| unsigned short int | 2 bytes | Stores fractional numbers, containing one or more decimals. Sufficient for storing 15 decimal digits | %usd |
| unsigned long int | 4 bytes | Stores a single character/letter/number, or ASCII values | %uld |

Here's how we make use of the data type of a variable to print it.

```c
#include <stdio.h>

int main()
{
    // Creating variables having different data types
```

```
    int integer = 26;
    float floating = 39.32;
    char character = 'A';

    // Printing variables with the help of their respective format specifiers
    printf("%d\n", integer);
    printf("%f\n", floating);
    printf("%c\n", character);
}
```

Output:

```
26
39.320000
A
```

## C Constants

- When you don't want the variables you declare to get modified intentionally or mistakenly in the later part of your program by you or others, you use the const keyword (this will declare the variable as "constant", which means unchangeable and read-only).
- You should always declare the variable as constant when you have values that are unlikely to change, like any mathematical constant as PI.
- When you declare a constant variable, it must be assigned a value.

Here's an example of how we declare a constant.

```
#include <stdio.h>

int main()
{
    const int MOD = 10000007;
}
```

---

# Chapter 7 - C Operators

Special symbols that are used to perform actions or operations are known as operators. They could be both unary or binary.

For example, the symbol asterisk (*) is used to perform multiplication in C so it is an operator and it is a binary operator.

This section covers all types of operators.

## Arithmetic Operators

Arithmetic operators are used to perform mathematical operations such as addition, subtraction, etc. A few of the simple arithmetic operators are

| Operator | Description |
|---|---|
| + | Addition |
| - | Subtraction |
| * | Multiplication |
| / | Division |
| % | Modulus |

We all must already know their purpose and how they are used in simple mathematics. Their purpose and functionality are the same.

Modulus(%) operator- this operator returns the remainder of two operands when they are been divided

```
#include<stdio>
int main()
{
int x;
x=5%2;
printf("remainder is %d",x);
}
```

Let's see their implementation in C.

```
#include <stdio.h>

int main()
{
    int a = 2;
    int b = 3;
    printf("a + b = %d\n", a + b);
}
```

Output:

```
a + b = 5
```

## Relational Operators

Relational operators are used for the comparison between two or more numbers or even expressions in cases. Same as Java, C also has six relational operators and their return value is of a Boolean type that is, either True or False (1 or 0).

| Operator | Description |
|---|---|
| > | Greater than |
| < | Less than |
| >= | Greater than or equal to |
| <= | Less than or equal to |
| == | Is equal to |
| != | Is not equal to |

Let's see their implementation in C.

```c
#include <stdio.h>

int main()
{
    int a = 2;
    int b = 3;
    printf("a == b = %d\n", a == b);
}
```

Output:

```
a == b = 0
```

The output is 0, since a and b are not equal.

## Logical Operators

There are three logical operators i.e. AND, OR, and NOT. They can be used to compare Boolean values but are mostly used to compare conditions to see whether they are satisfying or not.

- AND: it returns true when both operators are true or 1.
- OR: it returns true when either operator is true or 1.
- NOT: it is used to reverse the logical state of the operand.

| Operator | Description |
|---|---|
| && | AND Operator |
| \|\| | OR Operator |
| ! | NOT Operator |

Let's see their implementation in C.

```
#include <stdio.h>

int main()
{
    int a = 1;
    int b = 0;
    printf("a or b = %d\n", a || b);
}
```

Output:

```
a or b = 1
```

The output is 1, since either a or b is not equal to zero.

## Bitwise Operators

A bitwise operator is used to performing operations at the bit level. To obtain the results, they convert our input values into binary format and then process them using whatever operator they are being used with.

| Operator | Description |
|---|---|
| & | Bitwise AND |
| \| | Bitwise OR |
| ^ | Bitwise XOR |
| ~ | Bitwise Complement |
| >> | Shift Right Operator |
| << | Shift Left Operator |

Let's see their implementation in C.

```c
#include <stdio.h>

int main()
{
    int a = 2; //10
    int b = 3; //11
    printf("a xor b = %d\n", a ^ b);
}
```

Output:

```
a xor b = 1
```

The output is 1, since a xor b is 01 in binary, which is 1 in decimal.

## Assignment Operators

Assignment operators are used to assign values. We will use them in almost every program we develop.

```c
int a = 0;
int b = 1;
```

Equal to (=) is the assignment operator here. It is assigning 0 to a and 1 to b in the above example.

| Operator | Description |
|---|---|
| = | It assigns the right side operand value to the left side operand. |
| += | It adds the right operand to the left operand and assigns the result to the left operand. |
| -= | It subtracts the right operand from the left operand and assigns the result to the left operand. |
| *= | It multiplies the right operand with the left operand and assigns the result to the left operand. |
| /= | It divides the left operand with the right operand and assigns the result to the left operand. |

# Chapter 8 - Format Specifiers & Escape Sequences

Whenever we write a program in C, we have to use format specifiers to define the variable type in input and output and escape characters to format the output.

## Format Specifiers

A format specifier in C programming is used to define the type of data we are printing to the output or accepting through the input. Through this, we tell the compiler what type of variable we are using for input while using scanf() or output while using printf(). Some examples of format specifiers are %d, %c, %f, etc.

Here is a list of almost all format specifiers.

| Format Specifier | Type |
|---|---|
| %c | Used to print a character |
| %d | Used to print the signed integer |
| %f | Used to print the float values |
| %i | Used to print the unsigned integer |
| %l | Used to print the long integer |
| %lf | Used to print the double values |
| %lu | Used to print the unsigned integer or unsigned long integer |
| %s | Used to print the string |
| %u | Used to print the unsigned integer |

One example is shown below.

```c
#include <stdio.h>

int main()
{
    char c[100] = "My String";
    printf("Printing a string, %s.", c);
}
```

Output:

```
Printing a string, My String.
```

The %s used in the printf( ) is a format specifier. This format specifier tells printf( ) to consider it as a string and print accordingly.

## Escape Sequences

Many programming languages support the concept of Escape Sequences. An escape sequence is a sequence of characters that are used in formatting the output. They are not displayed on the screen while printing. Each character has its specific function. For example, \t is used to insert a tab, and \n is used to add a new line.

Here's the list of all escape sequences

| Escape Sequence | Description |
|---|---|
| \t | Inserts a tab space |
| \b | Inserts a backspace |
| \n | Inserts a new line |
| \r | Inserts a carriage return |
| \f | Inserts a form feed |
| \' | Inserts a single quote character |
| \" | Inserts a double quote character |
| \\ | Inserts a backslash character |

One example is shown below,

```c
#include <stdio.h>

int main()
{
    printf("Printing inside a double quotation, \"Hello World\"");
}
```

Output:

```
Printing inside a double quotation, “Hello World”.
```

---

# Chapter 9 - User Input/Output

We have already learned how the `printf()` function is used to output values in C. Another method, which goes by the name, `scanf()`, is used to get user input.

The `scanf()` function takes two arguments:

- the format specifier of the variable (as shown in the example below)
- the reference operator (&myNum), which stores the memory address of the variable. This is where the input data goes to.

## Syntax

```
scanf("format specifier",&variable_name);
```

- & - specifes the address of the variable.

One such example demonstrates how a program takes input from the user.

```
#include <stdio.h>

int main()
{
    int marks;
    char name[30];
    printf("Enter student's name: ");
    scanf("%s", name);
    printf("Enter marks in Maths: ");
    scanf("%d", &marks);

    printf("Hello %s! You have scored %d in Maths!", name, marks);
    return 0;
}
```

Input:

```
Enter student's name: Rohan
Enter marks in Maths: 98
```

Output:

```
Hello Rohan! You have scored 98 in Maths!
```

You must note that we didn't have to specify the reference operator (&) in cases of strings if we have specified the size of the strings already. This is an exception.

---

# Chapter 10 - C if...else statements

Sometimes, we wish to execute one set of instructions if a particular condition is met, and another set of instructions if it is not. This kind of situation is dealt with in C language using a decision control system.

The condition for the if statement is always enclosed within a pair of parentheses. If the condition is true, then the set of statements following the if statement will execute. And if the condition evaluates to false, then the statement will not execute, instead, the program skips that enclosed part of the code.

An expression in if statements are defined using relational operators. Comparing two values using relational operators allows us to determine whether they are equal, unequal, greater than, or less than.

If we want to execute a particular code in some situation and its vice versa /opposite/or different code if that situation doesn't occurs then if..else statements can be used. Its all depend on the condition. If the condition returns true value the situation has occurred and the true part of code will be executed and if condition returns false value false part of the code will be executed

| Conditions | Meaning |
|---|---|
| a==b | a is equal to b |
| a!=b | a is not equal to b |
| a<b | a is less than b |
| a>b | a is greater than b |
| a<=b | a is less than or equal to b |
| a>=b | a is greater than or equal to b |

The statement written in an if block will execute when the expression following if evaluates to true. But when the if block is followed by an else block, then when the condition written in the if block turns to be false, the set of statements in the else block will execute.

Following is the syntax of if-else statements:

```
if ( condition ){
statements;}
 else {
statements;}
```

One example where we could use the if-else statement is:

```
#include <stdio.h>

int main()
{
    int num = 10;
```

```
    if (num <= 10)
    {
        printf("Number is less than equal to 10.");
    }
    else
    {
        printf("Number is greater than 10.");
    }
    return 0;
}
```

Output:

```
Number is less than equal to 10.
```

## Ladder if-else

If we want to check the multiple conditions then ladder if else can used. If the previous condition returns false then only next condition will be checked.

syntax-

```
if(/*conditon*/)
{
//statements
}
else if(/*condition*/)
{
statements
}
else if(/*condition*/)
{
statements
}
```

## Nested if-else

We can also write an entire if-else statement within either the body of another if statement or the body of an else statement. This is called the 'nesting' of ifs.

```
if(/*condition*/)
{
if(/*condition*/)
{
//statements
}
else
```

```
{
//statements
}
}
else
{
//statements
}
```

---

# Chapter 11 - Switch Case Statements

## What is Switch?

The control statement that allows us to make a decision effectively from the number of choices is called a switch, or a switch case-default since these three keywords go together to make up the control statement. The expression in switch returns an integral value, which is then compared with different cases. Switch executes that block of code, which matches the case value. If the value does not match with any of the cases, then the default block is executed.

Following is the syntax of switch case-default statements:

```
switch ( integer expression )
{
case {value 1} :
do this ;

case {value 2} :
 do this ;

case {value 3} :
do this ;

default :
do this ;
 }
```

## Understanding the syntax:

The expression following the switch can be an integer expression or a character expression. The case value 1, and 2 are case labels that are used to identify each case individually. Remember, that case labels should be unique for each of the cases. If it is the same, it may create a problem while executing a program. At the end of the case labels, we always use a colon ( : ). Each case is associated with a block. A block contains multiple statements that are grouped together for a particular case.

Whenever the switch is executed, the value of test-expression is compared with all the cases present in switch statements. When the case is found, the block of statements associated with that particular case will execute. The break keyword indicates the end of a particular case. If we do not put the break in each case, then even though the specific case is executed, C's switch will continue to execute all the cases until the end is reached. The default case is optional. Whenever the expression's value is not matched with any of the cases inside the switch, then the default case will be executed.

One example where we could use the switch case statement is,

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

```c
int main()
{
    int i = 2;

    switch (i)
    {
    case 1:
        printf("Statement 1");
        break;

    case 2:
        printf("Statement 2");
        break;

    case 3:
        printf("Statement 3");
        break;

    default:
        printf("No valid i to switch to.");
        break;
    }
    return 0;
}
```

Output:

```
Statement 2
```

## Different to if-else. How?

There is one problem with the if statement: the program's complexity increases whenever the number of if statements increases. If we use multiple if-else statements in the program, the code might become difficult to read and comprehend. Sometimes it also even confuses the developer who himself wrote the program. Using the switch statement is the solution to this problem.

Furthermore,

- Switch statements cannot evaluate float conditions, and the test expression can only be an integer or a character, whereas if statements can evaluate float conditions as well.
- Switch statements cannot evaluate relational operators hence they are not allowed in switch statements, whereas if statements can evaluate relational operators.
- Cases in the switch can never have variable expressions; for example, we cannot write case a +3:

## Rules for Switch statements

- The test expression of Switch must necessarily be an int or char.
- The value of the case should be an integer or character.
- Cases should only be inside the switch statement.

- Using the break keyword in the switch statement is not necessary.
- The case label values inside the switch should be unique.

It is not necessary to use the break keyword after every case. Break keywords should only be used when we want to terminate our case at that time, otherwise we won't.

We can also use nested switch statements i.e., switch inside another switch. Also, the case constants of the inner and outer switch may have common values without any conflicts.

---

# Chapter 12 - C Loops

In programming, we often have to perform an action, again and again, with little or no variations in the details each time they are executed. This need is met by a mechanism known as a loop.

The versatility of the computer lies in its ability to perform the set of instructions repeatedly. This involves repeating some code in the program, either a specified number of times or until a particular condition is satisfied. Loop-controlled instructions are used to perform this repetitive operation efficiently ensuring the program doesn't look redundant at the same time due to the repetitions.

Following are the three types of loops in C programming.

- For loop
- While loop
- do-while loop

## Types of Loops

**Entry Controlled loops**

- In entry controlled loops, the test condition is evaluated before entering the loop body. The for Loop and the while Loop are examples of entry-controlled loops.

**Exit Controlled Loops**

- In exit-controlled loops, the test condition is tested at the end of the loop. Regardless of whether the test condition is true or false, the loop body will execute at least once. The do-while loop is an example of an exit-controlled loop.

## For Loop

A for loop is a repetition control structure that allows us to efficiently write a loop that will execute a specific number of times. The for loop working is as follows:

- The initialization statement is executed only once; in this statement, we initialize a variable to some value.
- In the second step, the test expression is evaluated. Suppose the test expression is evaluated to be true. In that case, the for loop keeps running, and the test expression is re-evaluated, but if the test expression is evaluated to false, then the for loop terminates.
- The loop keeps executing until the test expression is false. When the test expression is false, then the loop terminates.

## While loop

While loop is called a pre-tested loop. The while loop allows code to be executed multiple times, depending upon a boolean condition that is given as a test expression. While introducing for loops, we saw that the number of iterations is known, whereas while loops are used in situations where we do not know the exact

number of iterations of the loop. The while loop execution is terminated based on the test condition which evaluates to either true or false.

## do-while loop

In do-while loops, the execution is terminated based on the test condition, very similar to how it is done in a while loop. The main difference between the do-while loop and while loop is that, in the do-while loop, the condition is tested at the end of the loop body, whereas the other two loops are entry-controlled. In a do-while loop, the loop body will execute at least once irrespective of the test condition.

Sometimes, while executing a loop, it becomes necessary to jump out of the loop. For this, we make use of the break statement and the continue statement.

## break statement

Whenever a break statement is encountered in a loop, the loop is terminated regardless of what kind of loop we are in and the program continues with the statement following the loop.

## continue statement

Whenever a continue statement is encountered in a loop, it will cause the control to go directly to the test condition which is where the loop starts, ignoring any piece of code following the continue statement.

---

# Chapter 13 - while Loop

A While loop is also called a pre-tested loop. A while loop allows a piece of code in a program to be executed multiple times, depending upon a given test condition which evaluates to either true or false. The while loop is mostly used in cases where the number of iterations is not known. If the number of iterations is known, then we could also use a for loop.

The syntax for using a while loop,

```
while (condition test)
{
    // Set of statements
}
```

The body of a while loop can contain a single statement or a block of statements. The test condition may be any expression that should evaluate as either true or false. The loop iterates while the test condition evaluates to true.When the condition becomes false, the program control passes to the line immediately following the loop, which means, it terminates.

One such example to demonstrate how a while loop works is,

```
#include <stdio.h>

int main()
{
    int i = 0;
    while (i <= 5)
    {
        printf("%d ", i);
        i++;
    }
    return 0;
}
```

Output:

```
0 1 2 3 4 5
```

## Properties of the while loop:

Following are some of the properties of the while loop.

- A conditional expression written in the brackets of while is used to check the condition. The Set of statements defined inside the while loop will execute until the given condition returns false.

- The condition will return 0 if it is true. The condition will be false if it returns any nonzero number.
- In the while loop, we cannot execute the loop until we do not specify the condition expression.
- It is possible to execute a while loop without any statements. This will give no error.
- We can have multiple conditional expressions in a while loop.
- Braces are optional if the loop body contains only one statement.

---

# Chapter 14 - do-while Loop

A do-while loop is a little different from a normal while loop. A do-while loop, unlike what happens in a while loop, executes the statements inside the body of the loop before checking the test condition.

So even if a condition is false in the first place, the do-while loop would have already run once. A do-while loop is very much similar to a while loop, except for the fact that it is guaranteed to execute the body at least once.

Unlike for and while loops, which test the loop condition first, then execute the code written inside the body of the loop, the do-while loop checks its condition at the end of the loop.

Following is the syntax of the do-while loop.

```
do
{
    statements;
} while (test condition);
```

If the test condition returns true, the flow of control jumps back up to the do part, and the set of statements in the loop executes again. This process repeats until the given test condition becomes false.

## How does the do-while loop work?

- First, the body of the do-while loop is executed once. Only then, the test condition is evaluated.
- If the test condition returns true, the set of instructions inside the body of the loop is executed again, and the test condition is evaluated.
- The same process goes on until the test condition becomes false.
- If the test condition returns false, then the loop terminates.

One such example to demonstrate how a do-while loop works is

```
#include <stdio.h>

int main()
{
    int i = 5;

    do
    {
        printf("%d ", i);
        i++;
    } while (i < 5);

    return 0;
}
```

Output:

```
5
```

As it was already mentioned at the beginning of this tutorial, a do-while loop runs for at least once even if the test condition returns false, because the test condition is evaluated only after the first execution of the instructions in the body of the loop.

## Difference between a while and a do-while loop

A While loop is executed every time the given test condition returns true, whereas, a do-while loop is executed for the first time irrespective of the test condition being true or false because the test condition is checked only after executing the loop for the first time.

---

# Chapter 15 - for Loop

The "For" loop is used to repeat a specific piece of code in a program until a specific condition is satisfied. The for loop statement is very specialized. We use a for loop when we already know the number of iterations of that particular piece of code we wish to execute. Although, when we do not know about the number of iterations, we use a while loop.

Here is the syntax of a for loop in C programming.

```
for (initialise counter; test counter; increment / decrement counter)
{
    //set of statements
}
```

Here,

- initialize counter: It will initialize the loop counter value. It is usually i=0.
- test counter: This is the test condition, which if found true, the loop continues, otherwise terminates.
- Increment/decrement counter: Incrementing or decrementing the counter.
- Set of statements: This is the body or the executable part of the for loop or the set of statements that has to repeat itself.

One such example to demonstrate how a for loop works is,

```
#include <stdio.h>

int main()
{
    int num = 10;
    int i;
    for (i = 0; i < num; i++)
    {
        printf("%d ", i);
    }
    return 0;
}
```

Output:

```
0 1 2 3 4 5 6 7 8 9
```

- First, the initialization expression will initialize loop variables. The expression i=0 executes once when the loop starts. Then the condition i < num is checked. If the condition is true, then the statements inside the body of the loop are executed. After the statements inside the body are executed, the control

of the program is transferred to the increment of the variable i by 1. The expression i++ modifies the loop variables. Iteratively, the condition i < num is evaluated again.

- If the condition is still evaluated true, the body of the loop will execute once again. The for loop terminates when i finally becomes less than num, therefore, making the condition i<num false.
- Just as if statement, we can have a for loop inside another for loop. This is known as a nested for loop. Similarly, while loop and do while loop can also be nested.

## Why for loops if we already have while loops?

It is clear to a developer exactly how many times the loop will execute. So, if the developer has to dry run the code, it will become easier.

---

# Chapter 16 - C Break/Continue

## Break Statement

- Break statement is used to break the loop or switch case statements execution and brings the control to the next block of code after that particular loop or switch case it was used in.
- Break statements are used to bring the program control out of the loop it was encountered in.
- The break statement is used inside loops or switch statements in C Language.

One such example to demonstrate how a break statement works is,

```c
#include <stdio.h>

int main()
{
    int i = 0;
    while (1)
    {
        if (i > 5)
        {
            break;
        }
        printf("%d ", i);
        i++;
    }

    return 0;
}
```

Output:

```
0 1 2 3 4 5
```

Here, when i became 6, the break statement got executed and the program came out of the while loop.

## Continue Statement

- The continue statement is used inside loops in C Language. When a continue statement is encountered inside the loop, the control jumps to the beginning of the loop for the next iteration, skipping the execution of statements inside the body of the loop after the continue statement.
- It is used to bring the control to the next iteration of the loop.
- Typically, the continue statement skips some code inside the loop and lets the program move on with the next iteration.
- It is mainly used for a condition so that we can skip some lines of code for a particular condition.

- It forces the next iteration to follow in the loop unlike a break statement, which terminates the loop itself the moment it is encountered.

One such example to demonstrate how a continue statement works is

```
#include <stdio.h>

int main()
{
    for (int i = 0; i <= 10; i++)
    {
        if (i < 6)
        {
            continue;
        }
        printf("%d ", i);
    }
    return 0;
}
```

Output:

```
6 7 8 9 10
```

Here, the continue statement was continuously executing while i remained less than 5. For all the other values of i, we got the print statement working.

---

# Chapter 17 - Array Basics

An array is a collection of data items of the same data type. And it is also known as a subscript variable.

- Items are stored at contiguous memory locations in arrays.
- It can also store the collection of derived data types such as pointers, structures, etc.
- The C Language places no limits on the number of dimensions in an array. This means we can create arrays of any number of dimensions. It could be a 2D array or a 3D array or more.

## Dimensions of an array

- A one-dimensional array is like a list.
- A two-dimensional array is like a table.

Some texts refer to one-dimensional arrays as vectors and two-dimensional arrays as matrices and use the general term arrays when the number of dimensions is unspecified or unimportant.

## How do Arrays help?

Programs that use arrays for managing a large number of the same data type variables are more organized and readable in comparison to creating different variables of the same type for each data element.

Arrays allow us to create many variables by just a single line. It means there is no need to create or specify every variable.

## Advantages of Arrays?

- It is used to represent multiple data items of the same type by using only a single name.
- Accessing any random item at any random position in a given array is very fast in an array.
- There is no case of memory shortage or overflow in the case of arrays since the size is fixed and elements are stored in contiguous memory locations.

## Properties of Arrays

- An array stores data be it of any data type in contiguous memory locations in RAM.
- Each element of an array is of the same size i.e. their data types are the same so the memory consumed by each is also the same.
- Any element of the array with a given index can be accessed very quickly by using its address which can be calculated using the base address and the index number.

- **Index Number** – It is the special type of number which allows us to access variables of arrays. Index number provides a method to access each element of an array in a program.

# Chapter 18 - Array Operations

## Defining an array

**1. Without specifying the size of the array**

```
int arr[] = {1, 2, 3};
```

Here, we can leave the square brackets empty, although the array cannot be left empty in this case. It must have elements in it.

**2. With specifying the size of the array**

```
int arr[3];
arr[0] = 1, arr[1] = 2, arr[2] = 3;
```

Here, we can specify the size of the array in its definition itself. We can then put array elements later as well.

## Accessing an array element

An element in an array can easily be accessed through its index number. This must be remembered that the index number starts from 0 and not one.

Example:

```
#include <stdio.h>

int main()

{

    int arr[] = {1, 5, 7, 2};

    printf("%d ", arr[2]); //printing element on index 2

}
```

Output:

```
7
```

## Changing an array element

An element in an array can be overwritten using its index number.

Example:

```c
#include <stdio.h>

int main()

{

    int arr[] = {1, 5, 7, 2};

    arr[2] = 8; //changing the element on index 2

    printf("%d ", arr[2]); //printing element on index 2

}
```

Output:

```
8
```

---

# Chapter 19 - String Basics

## What are Strings?

A string is an array of characters. Data of the same type are stored in an array, for example, integers can be stored in an integer array, similarly, a group of characters can be stored in a character array. This character array is also known as strings. A string is a one-dimensional array of characters that is terminated by a null ('\0').

## Declaration of Strings:

Declaring a string is very simple, the same as declaring a one-dimensional array. It's just that we are considering it as an array of characters.

Below is the syntax for declaring a string.

```
char string_name[string_size];
```

In the above syntax, string_name is any name given to the string variable, and size is used to define the length of the string, i.e the number of characters that the strings will store. Keep in mind that there is an extra terminating character which is the null character ('\0') that is used to indicate the termination of the string.

Example of string:

```
#include <stdio.h>
int main()
{
    // declare and initialise string
    char str[] = "Hello World";
    printf("%s", str);
    return 0;
}
```

Output:

```
Hello World
```

# Chapter 20 - String Functions

We can use C's string handling library functions to handle strings. The string.h library is used to perform string operations. It provides several functions for manipulating strings.

Following are some commonly used string handling functions:

## 1. strcat():

This function is used to concatenate the source string to the end of the target string. This function expects two parameters, first, the base address of the source string and then the base address of the target string. For example, “Hello” and “World” on concatenation would result in a string “HelloWorld”.

Here is how we can use the strcat( ) function:

```c
#include <stdio.h>
#include <string.h>
int main()
{
    char s[] = "Hello";
    char t[] = "World";
    strcat(s, t);
    printf("String = %s", s);
}
```

Output:

```
String  = HelloWorld
```

## 2. strlen():

This function is used to count the number of characters present in a string.

Here is how we can use the strlen( ) function:

```c
#include <stdio.h>
#include <string.h>
int main()
{
    char s[] = "Hello";
    int len = strlen(s);
    printf("Length = %d", len);
}
```

Output:

```
Length  = 5
```

### 3. strcpy():

This function is used to copy the contents of one string into the other. This function expects two parameters, first, the base address of the source string and then the base address of the target string.

Here is how we can use the strcpy( ) function:

```
#include <stdio.h>
#include <string.h>
int main()
{
    char s[] = "Hello World";
    char t[50];
    strcpy(t, s);
    printf("Source string = %s\n", s);
    printf("Target string = %s", t);
}
```

Output: Source string = Hello World Target string = Hello World

### 4. strcmp():

The strcmp() function is used to compare two strings to find out whether they are the same or different. It takes two strings as two of its parameter. It will compare two strings, character by character until there is a mismatch or the iterator reaches the end of one of the strings.

If both of the strings are identical, strcmp( ) returns a value of zero. If they are not identical, it will return a value less than zero, considering the ASCII value of the mismatched character in the first string is less than the mismatched character in the second string. Else, it will return a value greater than 0.

Here is how we can use the strcmp( ) function:

```
#include <stdio.h>
#include <string.h>
int main()
{
    char s[] = "Hello";
    char t[] = "World";
    int cmp = strcmp(s, t);
    printf("Comparison result = %d", cmp);
}
```

Output:

```
Comparison result = -1
```

### 5. strrev():

This function is used to return the reverse of the string.

Here is how we can use the strrev( ) function:

```
#include <stdio.h>
#include <string.h>
int main()
{
    char s[] = "Hello";
    printf("Reversed string = %s", strrev(s));
}
```

Output:

```
Reversed string  = olleH
```

---

# Chapter 21 - Function Basics

## Functions:

- Functions are used to divide a large C program into smaller and less complex pieces.
- Function can be called multiple or several times to provide reusability and modularity to the C program.
- Functions are also called procedures or subroutines or methods.
- A function is also a piece of code that performs a specific task.

A function is nothing but a group of code put together and given a name and it can be called anytime without writing the whole code again and again in a program.

I know its syntax is a bit difficult to understand but don't worry after reading this whole information about Functions you will know each and every term or thing related to Functions.

## Advantages of Functions

- The use of functions allows us to avoid re-writing the same logic or code over and over again.
- With the help of functions, we can divide the work among the programmers.
- We can easily debug or can find bugs in any program using functions.
- They make code readable and less complex.

## Aspects of a function

**1. Declaration**

- This is where a function is declared to tell the compiler about its existence.

**2. Definition**

- A function is defined to get some task executed. (It means when we define a function, we write the whole code of that function and this is where the actual implementation of the function is done.)

**3. Call**

- This is where a function is called in order to be used.

## Types of functions

**1. Library functions:**

- Library functions are pre-defined functions in C Language. These are the functions that are included in C header files prior to any other part of the code in order to be used. E.g. printf(), scanf(), etc.

**2. User-defined functions**

- User-defined functions are functions created by the programmer for the reduction of the complexity of a program. Rather, these are functions that the user creates as per the requirements of a program. E.g. Any function created by the programmer.

# Chapter 22 - Function Parameters

A parameter or an argument of a function is the information we wish to pass to the function when it is called to be executed. 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.

Here's the basic syntax of a function

```
return_type functionName(parameter1, parameter2) {
  // body of the function
}
```

**What is the return_type?**

- Return type specifies the data type the function should return after its execution. It could also be a void where the function is required to return nothing.

## Different ways to define a function

**1. Without arguments and without return value**

In this function we don't have to give any value to the function (argument/parameters) and even don't have to take any value from it in return.

One example of such functions could be,

```
#include <stdio.h>

void func()
{
    printf("This function doesn't return anything.");
}

int main()
{
    func();
}
```

Output:

```
This function doesn't return anything.
```

**2. Without arguments and with the return value.**

In these types of functions, we don't have to give any value or argument to the function but in return, we get something from it i.e. some value.

One example of such functions could be,

```
#include <stdio.h>

int func()
{
    int a, b;
    scanf("%d", &a);
    scanf("%d", &b);
    return a + b;
}

int main()
{
    int ans = func();
    printf("%d", ans);
}
```

Input:

```
5
6
```

Output:

```
11
```

The function func when called, asks for two integer inputs and returns the sum of them. The function's return type is an integer.

**3. With arguments and without a return value.**

In this type of function, we give arguments or parameters to the function but we don't get any value from it in return.

One example of such functions could be,

```
#include <stdio.h>

void func(int a, int b)
```

```
{
    printf("%d", a * b);
}

int main()
{
    func(2, 3);
}
```

Output:

```
6
```

The function's return type is void. And it takes two integers as its parameters and prints the product of them while returning nothing.

**4. With arguments and with the return value**

In these functions, we give arguments to a function and in return, we also get some value from it.

One example of such functions could be,

```
#include <stdio.h>

int func(int a, int b)
{
    return a + b;
}

int main()
{
    int ans = func(2, 3);
    printf("%d", ans);
}
```

Output:

```
5
```

The function func takes two integers as its parameters and return an integer which is their sum.

> Make sure the return statement and the return_type of the functions are the same.

---

# Chapter 23 - Functions Declaration

**A function consist of two parts:**

- Declaration, where we specify the function's name, its return type, and required parameters (if any).
- Definition, which is nothing but the body of the function (code to be executed)

The basic structure of a function is,

```
return_type functionName() { // declaration
  // body of the function (definition)
}
```

For code optimization and readability, it is often recommended to separate the declaration and the definition of the function.

And this is the reason why you will often see C programs that have function declarations above the main() function, and function definitions below the main() function. This will make the code better organized and easier to read.

This is how it is done,

```
#include <stdio.h>

// Function declaration
void func();

int main()
{
    func(); // calling the function
    return 0;
}

// Function definition
void func()
{
    printf("This is the definition part.");
}
```

Output:

```
This is the definition part.
```

# Chapter 24 - Recursive Functions

## What are recursive functions?

Recursive functions or recursion is a process when a function calls a copy of itself to work on smaller problems.

Recursion is the process in which a function calls itself directly or indirectly. And the corresponding function or function which calls itself is called a recursive function.

- Any function which calls itself is called a recursive function.
- This makes the life of a programmer easy by dividing a complex problem into simple or easier problems.
- A termination condition is imposed on such functions to stop them from executing copies of themselves forever or infinitely. This is also known as the base condition.
- Any problem which can be solved recursively can also be solved iteratively.
- Recursions are used to solve tougher problems like Tower Of Hanoi, Fibonacci Series, Factorial finding, etc., and many more, where solving by intuition becomes tough.

## What is a base condition?

The case in which the function doesn't recur is called the base case.

For example, when we try to find the factorial of a number using recursion, the case when our number becomes smaller than 1 is the base case.

## Recursive Case

The instances where the function keeps calling itself to perform a subtask which is generally the same problem with the problem size reduced to many smaller parts, is called the recursive case.

Example of recursion,

```c
#include <stdio.h>

int factorial(int num)
{
    if (num > 0)
    {
        return num * factorial(num - 1);
    }
    else
    {
        return 0;
    }
}

int main()
{
```

```
    int ans = factorial(10);
    printf("%d", ans);
    return 0;
}
```

Output:

```
3628800
```

---

# Chapter 25 - C Pointers

When we initialise an array, we usually come to know about the,

- Memory block which is the space a variable gets in RAM. We can think of that space as a block.
- Name of the memory block which is the variable's name itself.
- Content of that block that is the value stored in that variable
- Address of the memory block assigned to the variable which is a unique address that allows us to access that variable.

## What is a Pointer?

- Pointer is a variable that contains the address of another variable. It means, it is a variable that points to any other variable.
- Although this is itself a variable, this contains the address or memory address of any other variable.
- It can be of type int, char, array, function, or even any other pointer.
- Its size depends on the architecture. Pointers in C Language can be declared using *(asterisk symbol). So, pointers are nothing but variables that store addresses of other variables, and by using pointers, we can access other variables too and can even manipulate them.

## Applications of Pointers

- Pointers are used to dynamically allocate or deallocate memory using methods such as malloc(), realloc(), calloc(), and free().
- Pointers are used to point to several containers such as arrays, or structs, and also for passing addresses of containers to functions.
- Return multiple values from a function
- Rather than passing a copy of a container to a function, we can simply pass its pointer. This helps reduce the memory usage of the program.
- Pointer reduces the code and improves the performance.

# Chapter 26 - Operations on Pointers

## Address of Operator (&):

- It is a unary operator.
- Operand must be the name of an already defined variable.
- & operator gives the address number of the variable.
- & is also known as the “Referencing Operator”.

Here’s one example to demonstrate the use of the address of the operator.

```
#include <stdio.h>

int main()
{
    int a = 100;
    printf("%d\n", a);
    printf("Address of variable a is %d", &a);
    return 0;
}
```

Output:

```
100
Address of variable a is 6422220
```

## Indirection Operator (*) :

- - is indirection operator.
- It is also known as the “Dereferencing Operator”.
- It is also a unary operator.
- It takes an address as an argument.
- - returns the content/container whose address is its argument.

Here’s one example to demonstrate the use of the indirection operator.

```
#include <stdio.h>

int main()
{
    int a = 100;
    printf("Value of variable a stored at address %d is %d.", &a, *(&a));
    return 0;
}
```

Output:

```
Value of variable a stored at address 6422220 is 100.
```

---

# Chapter 27 - C VOID Pointer

After a brief discussion about pointers, it's time to start with a very important type of pointers, void pointers and their functionalities. We already know that a void function has no return type i.e., functions that are not returning anything are given the type void. Now, in the case of pointers that are given the datatype of a void, they can be typecasted into any other data type according to the necessity. And that aids that we do not have to decide on a data type for the pointer initially.

Void pointers can also be addressed as general-purpose pointer variables.

Let's see a few examples that will demonstrate the functionalities of a void pointer.

Example:

```
int var = 1;
void *voidPointer = &var;
```

Here, the data type of the void pointer gets typecasted into int as we have stored the address of an integer value in it.

```
char x = 'a';
void *voidPointer = &x;
```

In this example, the void pointer's data type gets typecasted to char as we have stored the address of a character value in it.

Type casting a void pointer must also remind you of the way we used to type cast a void pointer returned by the functions malloc() and calloc() while for dynamic memory allocation. There also, the heap returns a void pointer to the memory requested. And we could type cast it to any other data type and that is where a void pointer comes handy.

## Two important features of a VOID pointer are:

**Void pointers cannot be dereferenced.** This can be demonstrated with the help of an example.

```
int a = 10;
void *voidPointer;
voidPointer = &a;
printf("%d", *voidPointer);
```

Output:

```
Compiler Error!
```

This program will throw a compile-time error, as we can not dereference a void pointer, meaning that we would compulsorily have to typecast the pointer every time it is being used. Here's how it should be done.

```
int a = 10;
void *voidPointer;
voidPointer = &a;
printf("%d", *(int *)voidPointer);
```

Output:

```
10
```

The compiler will not throw any error and will directly output the result because we are using the type along with the pointer.

> Pointer arithmetics cannot be used with void pointers since it is not holding any address to be able to increment or decrement its value.

---

# Chapter 28 - C NULL Pointer

A pointer that is not assigned any value or memory address but NULL is known as a NULL pointer. A NULL pointer does not point to any object, variable, or function. NULL pointers are often used to initialize a pointer variable, where we wish to represent that the pointer variable isn't currently assigned to any valid memory address yet.

This is how we define a NULL pointer,

```
int *ptr = NULL;
```

A NULL pointer generally points to a NULL or 0th memory location, so in simple words, no memory is allocated to a NULL pointer.

## Dereferencing a NULL pointer

The dereferencing behavior of a NULL pointer is very much similar to that of a void pointer. A NULL pointer itself is a kind of a VOID pointer and hence, we have to typecast it into any data type the way we do to a void pointer before dereferencing. Failing to do so results in an error at compile time.

## NULL pointer vs. Uninitialized pointer

NULL pointers and uninitialised pointers are different, as a Null pointer does not occupy any memory location. That means, it points to nowhere but to a zeroth location. In contrast, an uninitialized pointer means that the pointer occupies a garbage value address. The garbage value address is still a real memory location and hence not a NULL value. So to be on the safe side, NULL pointers are preferred.

## NULL pointer vs. Void pointer

NULL pointers and void pointers very much sound similar just because oif their nomenclatures, but they are very different as a NULL pointer is a pointer with a NULL value address, and a void pointer is a pointer of void data type. Their significances are contrasting.

An example of a NULL pointer is as follows,

```
int *ptr = NULL;
```

Here, an integer pointer variable is declared with a value NULL, which means it is not pointing to any memory location.

An example of a VOID pointer is as follows,

```
void *ptr;
```

Now, this is a void pointer. This pointer will typecast itself to any other data type as per the datatype of the value stored in it.

## Advantages of a NULL pointer

1. We can initialize a pointer variable without allocating any specific memory location to it.
2. We can use it to check whether a pointer is legitimate or not. We can check that by making the pointer a NULL pointer, after which it cannot be dereferenced.
3. A NULL pointer is used for comparison with other pointers to check whether that other pointer itself is pointing to some memory address or not.
4. We use it for error handling in the case of C programming.
5. We can pass a NULL pointer at places where we do not want to pass a pointer with a valid memory address.

---

# Chapter 29 - Dangling Pointer

Dangling pointers are pointers that are pointing to a memory location that has been already freed or deleted.

Dangling pointers often come into existence during object destruction. It happens when an object with an incoming reference is deleted or de-allocated, without modifying the value of the pointer. The pointer still points to the memory location of the deallocated memory.

The system may itself reallocate the previously deleted memory and several unpredicted results may occur as the memory may now contain different data.

## Dangling pointers are caused by the following factors:

**De-allocating or free variable memory**

When memory is deallocated, the pointer keeps pointing to freed space. An example to demonstrate how that happens is:

```
#include <stdio.h>
int main()
{
    int a = 80;
    int *ptr = (int *)malloc(sizeof(int));
    ptr = &a;
    free(ptr);
    return 0;
}
```

The above code demonstrates how a variable pointer *ptr and an integer variable a containing a value 80 was created. The pointer variable *ptr is created with the help of the malloc() function. As we know that malloc() function returns the void, so we use int * for type conversion to convert void pointer into int pointer.

**Function Call**

Now, we will see how the pointer becomes dangling with the function call.

```
#include <stdio.h>
int *myvalue()
{
    int a = 10;
    return &a;
}

int main()
{
    int *ptr = myvalue();
    printf("%d", *ptr);
```

```
    return 0;
}
```

Output:

```
Segmentation Fault!
```

In the above code, First, we create the main() function in which we have declared ptr pointer, which contains the return value of the func() function. When the function func() is called, the program control moves to the context of the int *func(). Then, the function func() returns the address of the integer variable a.

This is where the program control comes back to the main() function and the integer variable a becomes unavaialbale for the rest of the program execution. And the pointer ptr becomes dangling as it points to a memory location that has been freed or deleted from the stack. Hence, the program results in a segmentation fault.

Had this code been updated, and the integer variable been declared globally which is static and as we know, any static variable stores in global memory, the output would have been 10.

## How to avoid the Dangling pointer errors

The dangling pointer introduces nasty bugs into our programs, and these bugs often result in security holes. By merely initializing the pointer value to the NULL, these errors following the creation of dangling pointer can be avoided. After that, the pointer will no longer point to the freed memory location. While the reason behind assigning the NULL value to the pointer was to have the pointer not to point to any random or previously assigned memory location.

---

# Chapter 30 - Wild Pointer

Uninitialized pointers are known as wild pointers because they point to some arbitrary memory location while they are not assigned to any other memory location. This may even cause a program to crash or behave unpredictably at times.

For example:

```
int *ptr;
```

Here, a pointer named ptr is created but was not given any value. This makes the pointer ptr, a wild pointer.

Declaring a pointer and not initialising it has its own disadvantages. One such disadvantage is that it will store any garbage value in it. A random location in memory will be held in it arbitrarily. This random allocation often becomes tough for a programmer to debug causing a lot of problems in the execution of the program.

**A. Avoiding problems due to WILD pointers**

Dereferencing a wild pointer becomes problematic at times, and to avoid the them, we often prefer to convert a void pointer to a NULL pointer. By doing so, our pointer will not point to any garbage memory location, rather it will point to a NULL location. We can convert a wild pointer to a NULL pointer by merely putting it equal to NULL.

**B. Dereferencing**

We cannot dereference a wild pointer as we are not sure about the data in the memory it is pointing towards. In addition to causing a lot of bugs, dereferencing a wild pointer can also cause the program to crash.

# Chapter 31 - C Static Variables

## A. Local Variables

- Local variables are variables that are declared inside a function or a block of code. These variables cannot be accessed outside the function they are declared in. Local variables can be accessed only by statements that are inside that function or block of code which means the scope of these variables will be within the function only.

## B. Global Variables

- Global variables are variables that are defined outside of all the functions. Global variables hold their values, and we can access them inside any of the functions defined in the program. When we define global variables, the system automatically initializes them.
- If cases where we have both the local and global variables with the same name declared, the local variable takes preference.

## C. Static Variables

- A static variable is a variable that retains its value even after the program exits the scope it was declared in. Static variables retain their value and are not initialized again in the new scope. The memory assigned to a static variable stays until the end of the program is reached, whereas a normal variable is destroyed when a function it was declared in gets exited. They can be defined inside or outside the function. Static variables are local to the block. The default value of a static variable is zero. The keyword static is used to declare a static variable.

**Syntax:**

```
static datatype variable_name = variable_value;
```

## Difference between static local and static global variables

**Static global variable**

Any variable declared outside a function with a static keyword is known as a static global variable. This variable will be accessible through any of the methods in the program.

**Static local variable**

Any variable declared inside of a function with a static keyword is known as a static local variable. The scope of a static local variable will be the same as that of a local variable, but its memory will be available throughout the execution of the program.

## Properties of static variables

- A static variable will retain the value even after the program exits the scope it was declared in.
- Memory allocated to a static variable is available throughout the program execution.
- If we do not initialize a static variable, then the default value will be 0.

---

# Chapter 32 - C Memory Layout

## What is Dynamic Memory?

Any allocation of memory space during the runtime of the program is called dynamic memory allocation. The concept of dynamic memory allocation is used to reduce the wastage of memory, and it is the optimal way of memory allocation.

## Memory Allocation in C

Memory allocation in C can be divided into four segments.

**1. Code:**

Code composes of all the text segments of our program. Everything we do as a programmer to build a program falls into this category.

**2. Variables:**

Declarations of both global and static variables come into this segment. Global variables can be used anywhere in the program, while static has its limitations inside the function.

**3. Stack:**

A stack is a data structure. Initially, the stack looks like an empty bucket in which the last entry to be inserted will be the first one to get out. It is also known as a LIFO data structure i.e., last in first out.

Suppose the program starts executing a function named A, then this function A gets pushed into the stack. Now, if function A calls another function B during its execution, then function B will also get pushed into the stack, and the program will start executing B. Now, if B calls another function C, then the program will push C into the stack and will start with its execution. Now, after the program gets done with the execution of C, the program will pop C from the stack as it was the last one to get pushed and start executing B. When B gets executed completely, it will get popped and A will start executing further until the stack becomes empty.

**4. Heap:**

Heap is a tree-based data structure. It is used when we allocate memory dynamically. To use the heap data structure, we have to create a pointer in our main function that will point to some memory block in a heap. The disadvantage of using a heap is that the memory assigned to a pointer will not get freed automatically when the pointer gets overwritten.

## Differences between static and dynamic memory

| Static | Dynamic |
|---|---|
| Allocation of memory before execution | Allocation of memory at run time |
| Non-reusable memory | Reusable memory |

| Static | Dynamic |
| --- | --- |
| Less optimal way | More optimal way |

# Chapter 33 - C Memory Allocation

There are ways in which we can allocate memories dynamically in a heap. We use these four standard library functions often;

## 1. malloc():

malloc stands for memory allocation. This inbuilt function requests memory from the heap and returns a pointer to the memory. The pointer is of the void type and we can typecast it to any other data type of our choice.

All the values at the allocation time are initialized to garbage values. The function expects the memory space along with the size we want in bytes at the time it is used.

**Syntax:**

```
ptr = (ptr - type *)malloc(size_in_bytes)
```

Example:

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

## 2. calloc():

calloc stands for contiguous memory allocation. Similar to malloc, this function also requests memory from the heap and returns a pointer to the memory. Differences lie in the way we have to call it.

First, we have to send as parameters the number of blocks needed along with their size in bytes. Second, in calloc(), the values at the allocation time are initialized to 0 instead of garbage value unlike what happens in malloc().

**Syntax:**

```
ptr = (ptr - type *)calloc(n, size_in_bytes)
```

Example:

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

## 3. realloc():

realloc stands for reallocation of memory. It is used in cases where the dynamic memory allocated previously is insufficient and there is a need of increasing the already allocated memory to store more data.

We also pass the previously declared memory address, and the new size of the memory in bytes while calling the function.

**Syntax:**

```
ptr = (ptr - type *)realloc(ptr, new_size_in_bytes)
```

Example:

```
ptr = (int *)realloc(ptr,10* sizeof(int));
```

### 4. free():

While discussing the disadvantages of dynamic memory allocation, it was mentioned that there is no automatic deletion of dynamically allocated memory when the pointer gets overwritten. So, to manually do it, we use the free() function to free up the allocated memory space. Therefore, free() is used to free up the space occupied by the allocated memory.

We just have to pass the pointer as a parameter inside the function and the address being pointed gets freed.

**Syntax:**

```
free(ptr);
```

---

# Chapter 34 - C Structures

We have already learnt in the previous tutorials that variables store one piece of information and arrays of certain data types store the information of the same data type. Variables and arrays can handle a great variety of situations. But quite often, we have to deal with the collection of dissimilar data types. For dealing with cases where there is a requirement to store dissimilar data types, C provides a data type called 'structure'. It is a way to group together information belonging to different data types and combine them into one structure.

## What are Structures in C?

Structures are usually used when we wish to store data of different data types together. For example, if we want to store information about a book, there could be a number of parameters defining a book.

Books have a title, an author name, the number of pages, and a price. All of the book attributes belong to different data types. The titles and author names must be strings, but the prices and number of pages must be numerical.

One way to store the data is to construct individual arrays, and another method is to use a structure variable. It is to keep in mind that structure elements are always stored in contiguous memory locations.

## Creating a struct element

Basic syntax for declaring a struct is,

```
struct structure_name
{
    //structure_elements
} structure_variable;
```

Here's one example of how a struct is defined and used in main as a user-defined data type.

```
#include <stdio.h>

struct Books
{
    char title[20];
    char author[100];
    float price;
    int pages;
};

int main()
{
    struct Books book1;
    return 0;
}
```

## Accessing struct elements

We use the subscript operator, '[' fed with the index number to access individual elements of an array. But in the case of structures, to access any element, we use the dot operator (.). This dot operator is coded between the structure variable name and the structure member that we wish to access.

Before the dot operator, there must always be an already defined structure variable and after the dot operator, there must always be a valid structure element.

Here's one example demonstrating how we access struct elements.

```c
#include <stdio.h>

struct Books
{
    char title[20];
    char author[100];
    float price;
    int pages;
};

int main()
{
    struct Books book1 = {"C Programming", "ABC", 123.99, 300};
    printf("%s\n", book1.title);
    printf("%s\n", book1.author);
    printf("%f\n", book1.price);
    printf("%d\n", book1.pages);
    return 0;
}
```

Output:

```
C Programming
ABC
123.989998
300
```

## Additional Features of Structs

1. We can assign the values of a structure variable to another structure variable of the same type using the assignment operator.
2. Structure can be nested within another structure which means structures can have their members as structures themselves.
3. We can pass the structure variable to a function. We can pass the individual structure elements or the entire structure variable into the function as an argument. And functions can also return a structure variable.

4. We can have a pointer pointing to a struct just like the way we can have a pointer pointing to an int, or a pointer pointing to a char variable.

## Where are Structs useful?

It is possible to use structures for many different purposes, including:

1. Structures are used to store a large amount of data of varying data types.
2. They are used to send data to the printer.
3. For placing the cursor at an appropriate position on the screen, we can use structure.
4. It can be used in drawing and floppy formatting.
5. We use structures in finding out the list of equipment attached to the computer.

---

# Chapter 35 - C Unions

Just like Structures, the union is a user-defined data type. All the members in unions share the same memory location. The union is a data type that allows different data belonging to different data types to be stored in the same memory locations. One of the advantages of using a union is that it provides an efficient way of reusing the memory location, as only one of its members can be accessed at a time. A union is used in the same way we declare and use a structure. The difference lies just in the way memory is allocated to their members.

## Defining a Union

We use the union keyword to define the union.

The syntax for defining a union is,

```
union union_name
{
    //union_elements
} structure_variable;
```

Here's one example of how a union is defined and used in main as a user defined data type.

```
#include <stdio.h>

union Books
{
    char title[20];
    char author[100];
    float price;
    int pages;
};

int main()
{
    union Books book1;
    return 0;
}
```

## Initialising and accessing union elements

Different from how we used to initialise a struct in one single statement, union elements are initialised one at a time.

And also, one can access only one union element at a time. Altering one union element disturbs the value stored in other union elements.

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

union Books
{
    char title[20];
    char author[100];
    float price;
    int pages;
};

int main()
{
    union Books book1;
    strcpy(book1.title, "C Programming");
    printf("%s\n", book1.title);

    strcpy(book1.author, "ABC");
    printf("%s\n", book1.author);

    book1.price = 123.99;
    printf("%f\n", book1.price);

    book1.pages = 300;
    printf("%d\n", book1.pages);

    return 0;
}
```

Output:

```
C Programming
ABC
123.989998
300
```

## How are Structs and Unions similar?

1. Structures and unions, both are user-defined data types used to store data of different types.
2. The members of structures and unions can be objects of any type, including even other structures and unions or arrays.
3. A union or a structure can be passed by value to functions and can be returned by value by functions.
4. '.' operator is used for accessing both union and structure members.

## How are Structs and Unions different?

1. The keyword union is used to define a union and a keyword struct is used to define the structure

2. Within a structure, each member is allocated a unique storage area of location whereas memory allocated to a union is shared by individual members of the union.
3. Individual members can be accessed at a time in structures whereas only one member can be accessed at a time in unions.
4. Changing the value of one of the members of a structure will not affect the values of the other members of the structure, whereas changing the value of one of the members of a union will affect the values of other members in a union.
5. Several members of a structure can be initialised at once, whereas only one member can be initialised in the union.

---

# Chapter 36 - C Typedef

In C programming, a typedef declaration is used to create shorter and more meaningful and convenient to use names for keywords already defined by C like int, float, and char.

## What is a typedef in C?

A typedef is a keyword that is used to assign alternative names to existing datatypes. We use typedef with user defined datatypes, when the names of the datatypes become slightly complicated to use in programs. Typedefs can be used to:

- Make a more complex definition reusable by abbreviating it to something less complex.
- Provide more clarity to the code
- Make it easier to change the underlying data types that we use
- Make the code more clear and easier to modify.

Following is the syntax for using typedef,

```
typedef <previous_name> <alias_name>
```

For example, we would often want to create a variable of type unsigned long. But, then it becomes a complex task if that need to declare an unsigned long comes for multiple variables. To overcome this problem, we use a typedef keyword.

Here is how we use it.

```
#include <stdio.h>
typedef unsigned long ul;

int main()
{
    ul a;
}
```

## Applications of Typedef

There are various applications of typedef. Listed below are a few applications of typedef.

- The typedef can be used with arrays, primarily multi-dimensional arrays. It increase the readability of the program.
- Typedefs can also be implemented for defining a user-defined data type like structs or unions with a specific name and type.

Here's how we use typedefs for defining a struct in C.

```
typedef struct
{
    structure element1;
    structure element2;
    structure element3;
} name_of_type
```

typedef can be used for providing a pseudo name to pointer variables as well.

```
typedef int *ptr;
```

## Advantages of Typedef

- Typedef, as mentioned, increases the readability of the code. If we are using structure or function pointer or long keywords repeatedly in our code, then using typedefs increases the readability of code.
- With the help of typedef, we can use the same name for different applications even in different scopes.
- In the case of structure, if we use the typedef then we do not require to write struct keyword at the time of variable declaration.
- Typedef increases the portability of the code.

---

# Chapter 37 - File Handling Basics

A large program deployed for heavy applications cannot function without files, since we can get input from them as well as print output from them very easily. We can also save a lot of program space by accessing the file's data only when needed, making the program more efficient and faster.

Why do we need to handle files in C?

- Files are used to store content hence reducing the actual program's size.
- We can read or access data from files.
- The data in files remain stored even after the program's execution is terminated.

Files are stored in non-volatile memory. To understand what a non-volatile memory is and how it is better in terms of storing things for longer, we have to see the differences between volatile and non-volatile memory.

| **VOLATILE MEMORY** | **NON- VOLATILE MEMORY** |
|---|---|
| The data can only remain in the system while the computer’s power is on. | The data remains in the program even after the computer’s power is off. It gets retrieved after the system gets on again. |
| Volatile memory can only hold information when there is a constant power supply. | Non volatile memory can also hold information even in case of non constant power supply. |
| Data gets held for a short period in case of volatile memory. | Data gets held for a longer term in case of non-volatile memory. |
| RAM is an example | Hard Disk is an example. |

## What are the different type of files?

There are two types of files:

**Binary Files**

- Binary files store data in 01 i.e., binary format. They are not directly readable. In order to read binary files, you will need software or an application. An example of binary file is a .bin file.

**Text Files**

- Text files store data in a simple text format. They are directly readable and no external software is required to access them. An example of a text file is a .txt file.

# Chapter 38 - Operations on Files

There are basically four operations we can perform on files in C:

## Creating a File:

We can create a file using C language, in any directory, without even leaving our compiler. We can select the name or type we want our file to have, along with its location.

## Opening a File:

We can open an existing file and create a new file and open it using our program. We can perform different operations on a file after it has been opened.

## Closing a File:

When we are done with the file, meaning that we have performed whatever we want to perform on our file, we can close the file using the close function.

## Read/Write to a file:

After opening a file, we can access its contents and read, write, or update them.

# Chapter 39 - Files I/O

The first and foremost thing we should know when working with files in C is that we have to declare a pointer of the file type to work with files. The syntax for declaring a pointer of file type is

```
FILE *ptr;
```

## Modes

Functions and their modes of declarations are two important factors of file handling. We have to learn about different modes used along with these functions as a parameter. The following are the modes:

- `r`: opens a file for reading.
- `w`: opens a file for writing. It can also create a new file.
- `a`: opens a file for appending.
- `r+`: opens a file for both reading and writing but cannot create a new file.
- `w+`: opens a file for both reading and writing. There are many other modes, but these are the basic and most used ones.

## Closing a file

When working with C, closing open files is an essential step. A programmer often makes a mistake of not closing an open file. This becomes crucial because files do not automatically get closed after a program uses them. The closing has to be done manually.

To close a file, we have to use the fclose() function. We only need to pass the pointer as a parameter to the function.

**Syntax:**

```
fclose(ptr);
```

## Reading a file

Reading from a file is as easy as reading any other stuff as an input in C. We just use a file version of scanf(). In order to read from a file, we use the function fscanf(). Like scanf() used to get input from the keyboard, it gets its input from a file and prints it onto the screen.

We have to send the file pointer as an argument for the program to be able to read it. The file has to be opened in r mode, i.e., read mode, to work properly for fscanf().

Example:

```

```

```c
#include <stdio.h>

int main()
{
    FILE *ptr;
    ptr = fopen("example.txt", "r");
    char str[128];
    fscanf(ptr, "%s", str);
    printf("%s", str);
}
```

The file example.txt had “Hello World!” as its content, hence the output:

```
Hello World!
```

## Writing to a file

Writing to a file is as easy as printing any other stuff in C. We just use a file version of printf(). In order to write to a file, we use the function fprintf(). For printing text inside the file, we use fprintf() as we did for printing text on the screen using printf(). We have to send the file pointer as an argument for the program to be able to print it into the file. The file has to be opened in w mode, i.e. write mode, to be able to write in the file properly.

fprintf() takes the pointer to the file as one of the arguments along with the text to be written.

Example:

```c
#include <stdio.h>

int main()
{
    FILE *ptr;
    ptr = fopen("example.txt", "w");
    char str[128] = "Hello World!";
    fprintf(ptr, "%s", str);
}
```

Output in the example.txt file:

```
Hello World!
```

---