**Module - 1 (Computer Basics)**

**2. Programming Basics with C**

**C language** Tutorial with programming approach for beginners and professionals, helps you to understand the C language tutorial easily. Our C tutorial explains each topic with programs.

The C Language is developed for creating system applications that directly interact with the hardware devices such as drivers, kernels, etc.

C programming is considered as the base for other programming languages, that is why it is known as mother language.

It can be defined by the following ways:

1. Mother language
2. System programming language
3. Procedure-oriented programming language
4. Structured programming language
5. Mid-level programming language

1) C as a mother language

C language is considered as the mother language of all the modern programming languages because most of the compilers, JVMs, Kernels, etc. are written in C language, and most of the programming languages follow C syntax, for example, C++, Java, C#, etc.

It provides the core concepts like the array, strings, functions, file handling, etc. that are being used in many languages like C++, Java, C#, etc.

2) C as a system programming language

A system programming language is used to create system software. C language is a system programming language because it can be used to do low-level programming (for example driver and kernel). It is generally used to create hardware devices, OS, drivers, kernels, etc. For example, Linux kernel is written in C.

It can't be used for internet programming like Java, .Net, PHP, etc.

3) C as a procedural language

A procedure is known as a function, method, routine, subroutine, etc. A procedural language specifies a series of steps for the program to solve the problem.

A procedural language breaks the program into functions, data structures, etc.

C is a procedural language. In C, variables and function prototypes must be declared before being used.

4) C as a structured programming language

A structured programming language is a subset of the procedural language. Structure means to break a program into parts or blocks so that it may be easy to understand.

In the C language, we break the program into parts using functions. It makes the program easier to understand and modify.

5) C as a mid-level programming language

C is considered as a middle-level language because it supports the feature of both low-level and high-level languages. C language program is converted into assembly code, it supports pointer arithmetic (low-level), but it is machine independent (a feature of high-level).

A Low-level language is specific to one machine, i.e., machine dependent. It is machine dependent, fast to run. But it is not easy to understand.

A High-Level language is not specific to one machine, i.e., machine independent. It is easy to understand.

C Program

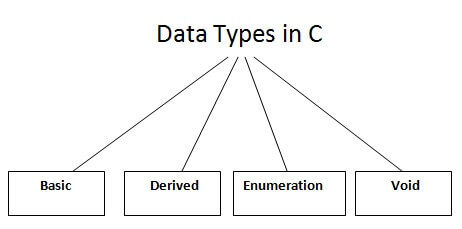
In this tutorial, all C programs are given with C compiler so that you can quickly change the C program code.

File: main.c

1. #include <stdio.h>
2. **int** main() {
3. printf("Hello C Programming\n");
4. **return** 0;
5. }

# Data Types in C

A data type specifies the type of data that a variable can store such as integer, floating, character, etc.



There are the following data types in C language.

|  |  |
| --- | --- |
| **Types** | **Data Types** |
| Basic Data Type | int, char, float, double |
| Derived Data Type | array, pointer, structure, union |
| Enumeration Data Type | enum |
| Void Data Type | void |

The memory size of the basic data types may change according to 32 or 64-bit operating system.

Let's see the basic data types. Its size is given **according to 32-bit architecture**.

|  |  |  |
| --- | --- | --- |
| **Data Types** | **Memory Size** | **Range** |
| **char** | 1 byte | −128 to 127 |
| signed char | 1 byte | −128 to 127 |
| unsigned char | 1 byte | 0 to 255 |
| **short** | 2 byte | −32,768 to 32,767 |
| signed short | 2 byte | −32,768 to 32,767 |
| unsigned short | 2 byte | 0 to 65,535 |
| **int** | 2 byte | −32,768 to 32,767 |
| signed int | 2 byte | −32,768 to 32,767 |
| unsigned int | 2 byte | 0 to 65,535 |
| **short int** | 2 byte | −32,768 to 32,767 |
| signed short int | 2 byte | −32,768 to 32,767 |
| unsigned short int | 2 byte | 0 to 65,535 |
| **long int** | 4 byte | -2,147,483,648 to 2,147,483,647 |
| signed long int | 4 byte | -2,147,483,648 to 2,147,483,647 |
| unsigned long int | 4 byte | 0 to 4,294,967,295 |
| **float** | 4 byte |  |
| **double** | 8 byte |  |
| **long double** | 10 byte |  |

# C Operators

An operator is simply a symbol that is used to perform operations. There can be many types of operations like arithmetic, logical, bitwise, etc.

There are following types of operators to perform different types of operations in C language.

* Arithmetic Operators
* Relational Operators
* Shift Operators
* Logical Operators
* Bitwise Operators
* Ternary or Conditional Operators
* Assignment Operator
* Misc Operator

## Precedence of Operators in C

The precedence of operator species that which operator will be evaluated first and next. The associativity specifies the operator direction to be evaluated; it may be left to right or right to left.

Let's understand the precedence by the example given below:

1. **int** value=10+20\*10;

The value variable will contain **210** because \* (multiplicative operator) is evaluated before + (additive operator).

The precedence and associativity of C operators is given below:

|  |  |  |
| --- | --- | --- |
| **Category** | **Operator** | **Associativity** |
| Postfix | () [] -> . ++ - - | Left to right |
| Unary | + - ! ~ ++ - - (type)\* & sizeof | Right to left |
| Multiplicative | \* / % | Left to right |
| Additive | + - | Left to right |
| Shift | << >> | Left to right |
| Relational | < <= > >= | Left to right |
| Equality | == != | Left to right |
| Bitwise AND | & | Left to right |
| Bitwise XOR | ^ | Left to right |
| Bitwise OR | | | Left to right |
| Logical AND | && | Left to right |
| Logical OR | || | Left to right |
| Conditional | ?: | Right to left |
| Assignment | = += -= \*= /= %=>>= <<= &= ^= |= | Right to left |
| Comma | , | Left to right |

# C Input Output (I/O)

**In this tutorial, you will learn to use scanf() function to take input from the user, and printf() function to display output to the user.**

## C Output

In C programming, printf() is one of the main output function. The function sends formatted output to the screen. For example,

### Example 1: C Output

1. #include <stdio.h>
2. int main()
3. {
4. // Displays the string inside quotations
5. printf("C Programming");
6. return 0;
7. }

**Output**

C Programming

How does this program work?

* All valid C programs must contain the main() function. The code execution begins from the start of the main() function.
* The printf() is a library function to send formatted output to the screen. The function prints the string inside quotations.
* To use printf() in our program, we need to include stdio.h header file using #inclue <stdio.h> statement.
* The return 0; statement inside the main() function is the "Exit status" of the program. It's optional.

### Example 2: Integer Output

1. #include <stdio.h>
2. int main()
3. {
4. int testInteger = 5;
5. printf("Number = %d", testInteger);
6. return 0;
7. }

**Output**

Number = 5

We use %d format specifier to print int types. Here, the %d inside the quotations will be replaced by the value of testInteger.

### Example 3: float and double Output

1. #include <stdio.h>
2. int main()
3. {
4. float number1 = 13.5;
5. double number2 = 12.4;
6. printf("number1 = %f\n", number1);
7. printf("number2 = %lf", number2);
8. return 0;
9. }

**Output**

number1 = 13.500000

number2 = 12.400000

To print float, we use %f format specifier. Similarly, we use %lf to print double values.

### Example 4: Print Characters

1. #include <stdio.h>
2. int main()
3. {
4. char chr = 'a';
5. printf("character = %c.", chr);
6. return 0;
7. }

**Output**

character = a

To print char, we use %c format specifier.

## C Input

In C programming, scanf() is one of the commonly used function to take input from the user. The scanf() function reads formatted input from the standard input such as keyboards.

### Example 5: Integer Input/Output

1. #include <stdio.h>
2. int main()
3. {
4. int testInteger;
5. printf("Enter an integer: ");
6. scanf("%d", &testInteger);
7. printf("Number = %d",testInteger);
8. return 0;
9. }

**Output**

Enter an integer: 4

Number = 4

Here, we have used %d format specifier inside the scanf() function to take int input from the user. When the user enters an integer, it is stored in the testInteger variable.

Notice, that we have used &testInteger inside scanf(). It is because &testInteger gets the address of testInteger, and the value entered by the user is stored in that address.

### Example 6: Float and Double Input/Output

1. #include <stdio.h>
2. int main()
3. {
4. float num1;
5. double num2;
6. printf("Enter a number: ");
7. scanf("%f", &num1);
8. printf("Enter another number: ");
9. scanf("%lf", &num2);
10. printf("num1 = %f\n", num1);
11. printf("num2 = %lf", num2);
12. return 0;
13. }

**Output**

Enter a number: 12.523

Enter another number: 10.2

num1 = 12.523000

num2 = 10.200000

We use %f and %lf format specifier for float and double respectively.

### Example 7: C Character I/O

1. #include <stdio.h>
2. int main()
3. {
4. char chr;
5. printf("Enter a character: ");
6. scanf("%c",&chr);
7. printf("You entered %c.", chr);
8. return 0;
9. }

**Output**

Enter a character: g

You entered g.

When a character is entered by the user in the above program, the character itself is not stored. Instead, an integer value (ASCII value) is stored.

And when we display that value using %c text format, the entered character is displayed. If we use %d to display the character, it's ASCII value is printed.

### Example 8: ASCII Value

1. #include <stdio.h>
2. int main()
3. {
4. char chr;
5. printf("Enter a character: ");
6. scanf("%c", &chr);
7. // When %c is used, a character is displayed
8. printf("You entered %c.\n",chr);
9. // When %d is used, ASCII value is displayed
10. printf("ASCII value is % d.", chr);
11. return 0;
12. }

Output

Enter a character: g

You entered g.

ASCII value is 103.

## I/O Multiple Values

Here's how you can take multiple inputs from the user and display them.

1. #include <stdio.h>
2. int main()
3. {
4. int a;
5. float b;
6. printf("Enter integer and then a float: ");
8. // Taking multiple inputs
9. scanf("%d%f", &a, &b);
10. printf("You entered %d and %f", a, b);
11. return 0;
12. }

**Output**

Enter integer and then a float: -3

3.4

You entered -3 and 3.400000

## Format Specifiers for I/O

As you can see from the above examples, we use

* %d for int
* %f for float
* %lf for double
* %c for char

Here's a list of commonly used C data types and their format specifiers.

| Data Type | Format Specifier |
| --- | --- |
| int | %d |
| char | %c |
| float | %f |
| double | %lf |
| short int | %hd |
| unsigned int | %u |
| long int | %li |
| long long int | %lli |
| unsigned long int | %lu |
| unsigned long long int | %llu |
| signed char | %c |
| unsigned char | %c |
| long double | %Lf |

# C if...else Statement

**In this tutorial, you will learn about if statement (including if...else and nested if..else) in C programming with the help of examples.**

## C if Statement

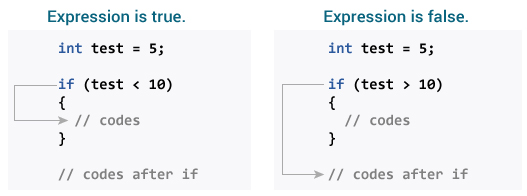
The syntax of the if statement in C programming is:

1. if (test expression)
2. {
3. // statements to be executed if the test expression is true
4. }

### How if statement works?

The if statement evaluates the test expression inside the parenthesis ().

* If the test expression is evaluated to true, statements inside the body of if are executed.
* If the test expression is evaluated to false, statements inside the body of if are not executed.



To learn more about when test expression is evaluated to true (non-zero value) and false (0), check [relational](https://www.programiz.com/c-programming/c-operators#relational) and [logical operators](https://www.programiz.com/c-programming/c-operators#logical).

### Example 1: if statement

1. // Program to display a number if it is negative
2. #include <stdio.h>
3. int main()
4. {
5. int number;
6. printf("Enter an integer: ");
7. scanf("%d", &number);
8. // true if number is less than 0
9. if (number < 0)
10. {
11. printf("You entered %d.\n", number);
12. }
13. printf("The if statement is easy.");
14. return 0;
15. }

**Output 1**

Enter an integer: -2

You entered -2.

The if statement is easy.

When the user enters -2, the test expression number<0 is evaluated to true. Hence, You entered -2 is displayed on the screen.

**Output 2**

Enter an integer: 5

The if statement is easy.

When the user enters 5, the test expression number<0 is evaluated to false and the statement inside the body of if is not executed

## C if...else Statement

The if statement may have an optional else block. The syntax of the if..else statement is:

1. if (test expression) {
2. // statements to be executed if the test expression is true
3. }
4. else {
5. // statements to be executed if the test expression is false
6. }

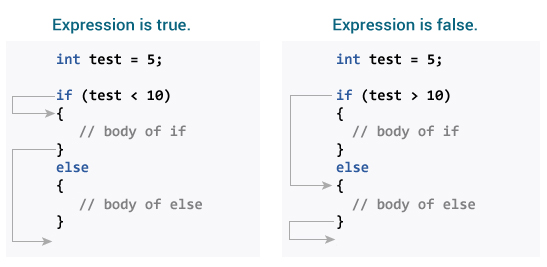
### How if...else statement works?

If the test expression is evaluated to true,

* statements inside the body of if are executed.
* statements inside the body of else are skipped from execution.

If the test expression is evaluated to false,

* statements inside the body of else are executed
* statements inside the body of if are skipped from execution.



### Example 2: if...else statement

1. // Check whether an integer is odd or even
2. #include <stdio.h>
3. int main()
4. {
5. int number;
6. printf("Enter an integer: ");
7. scanf("%d", &number);
8. // True if the remainder is 0
9. if (number%2 == 0)
10. {
11. printf("%d is an even integer.",number);
12. }
13. else
14. {
15. printf("%d is an odd integer.",number);
16. }
17. return 0;
18. }

**Output**

Enter an integer: 7

7 is an odd integer.

When the user enters 7, the test expression number%2==0 is evaluated to false. Hence, the statement inside the body of else is executed.

## C if...else Ladder

The if...else statement executes two different codes depending upon whether the test expression is true or false. Sometimes, a choice has to be made from more than 2 possibilities.

The if...else ladder allows you to check between multiple test expressions and execute different statements.

### Syntax of nested if...else statement.

1. if (test expression1)
2. {
3. // statement(s)
4. }
5. else if(test expression2)
6. {
7. // statement(s)
8. }
9. else if (test expression3)
10. {
11. // statement(s)
12. }
13. .
14. .
15. else
16. {
17. // statement(s)
18. }

### Example 3: C if...else Ladder

1. // Program to relate two integers using =, > or < symbol
2. #include <stdio.h>
3. int main()
4. {
5. int number1, number2;
6. printf("Enter two integers: ");
7. scanf("%d %d", &number1, &number2);
8. //checks if the two integers are equal.
9. if(number1 == number2)
10. {
11. printf("Result: %d = %d",number1,number2);
12. }
13. //checks if number1 is greater than number2.
14. else if (number1 > number2)
15. {
16. printf("Result: %d > %d", number1, number2);
17. }
18. //checks if both test expressions are false
19. else
20. {
21. printf("Result: %d < %d",number1, number2);
22. }
23. return 0;
24. }

**Output**

Enter two integers: 12

23

Result: 12 < 23

## Nested if...else

It is possible to include an if...else statement inside the body of another if...else statement.

### Example 4: Nested if...else

This program given below relates two integers using either <, > and = similar to the if...else ladder's example. However, we will use a nested if...else statement to solve this problem.

1. #include <stdio.h>
2. int main()
3. {
4. int number1, number2;
5. printf("Enter two integers: ");
6. scanf("%d %d", &number1, &number2);
7. if (number1 >= number2)
8. {
9. if (number1 == number2)
10. {
11. printf("Result: %d = %d",number1,number2);
12. }
13. else
14. {
15. printf("Result: %d > %d", number1, number2);
16. }
17. }
18. else
19. {
20. printf("Result: %d < %d",number1, number2);
21. }
22. return 0;
23. }

If the body of an if...else statement has only one statement, you do not need to use brackets {}.

For example, this code

1. if (a > b) {
2. print("Hello");
3. }
4. print("Hi");

is equivalent to

1. if (a > b)
2. print("Hello");
3. print("Hi");

# C for Loop

**In this tutorial, you will learn to create for loop in C programming with the help of examples.**

In programming, loops are used to repeat a block of code until a specified condition is met.

C programming has three types of loops:

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

We will learn about for loop in this tutorial. In the next tutorial, we will learn about while and do...while loop.

## for Loop

The syntax of the for loop is:

1. for (initializationStatement; testExpression; updateStatement)
2. {
3. // statements inside the body of loop
4. }

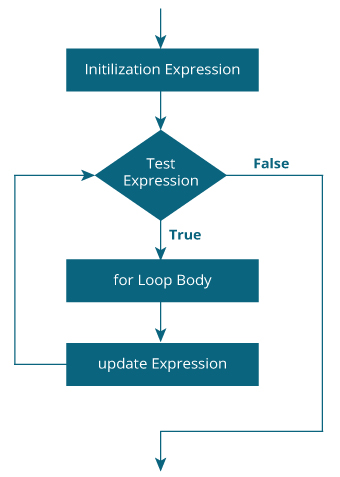
### How for loop works?

* The initialization statement is executed only once.
* Then, the test expression is evaluated. If the test expression is evaluated to false, the for loop is terminated.
* However, if the test expression is evaluated to true, statements inside the body of for loop are executed, and the update expression is updated.
* Again the test expression is evaluated.

This process goes on until the test expression is false. When the test expression is false, the loop terminates.

To learn more about test expression (when the test expression is evaluated to true and false), check out [relational](https://www.programiz.com/c-programming/c-operators#relational) and [logical operators](https://www.programiz.com/c-programming/c-operators#logical).

### for loop Flowchart



### Example 1: for loop

1. # Print numbers from 1 to 10
2. #include <stdio.h>
3. int main() {
4. int i;
5. for (i = 1; i < 11; ++i)
6. {
7. printf("%d ", i);
8. }
9. return 0;
10. }

**Output**

1 2 3 4 5 6 7 8 9 10

1. i is initialized to 1.
2. The test expression i < 11 is evaluated. Since 1 less than 11 is true, the body of for loop is executed. This will print the **1** (value of i) on the screen.
3. The update statement ++i is executed. Now, the value of i will be 2. Again, the test expression is evaluated to true, and the body of for loop is executed. This will print **2** (value of i) on the screen.
4. Again, the update statement ++i is executed and the test expression i < 11 is evaluated. This process goes on until i becomes 11.
5. When i becomes 11, i < 11 will be false, and the for loop terminates.

### Example 2: for loop

1. // Program to calculate the sum of first n natural numbers
2. // Positive integers 1,2,3...n are known as natural numbers
3. #include <stdio.h>
4. int main()
5. {
6. int num, count, sum = 0;
7. printf("Enter a positive integer: ");
8. scanf("%d", &num);
9. // for loop terminates when num is less than count
10. for(count = 1; count <= num; ++count)
11. {
12. sum += count;
13. }
14. printf("Sum = %d", sum);
15. return 0;
16. }

**Output**

Enter a positive integer: 10

Sum = 55

The value entered by the user is stored in the variable num. Suppose, the user entered 10.

The count is initialized to 1 and the test expression is evaluated. Since the test expression count<=num (1 less than or equal to 10) is true, the body of for loop is executed and the value of sum will equal to 1.

Then, the update statement ++count is executed and the count will equal to 2. Again, the test expression is evaluated. Since 2 is also less than 10, the test expression is evaluated to true and the body of for loop is executed. Now, the sum will equal 3.

This process goes on and the sum is calculated until the count reaches 11.

When the count is 11, the test expression is evaluated to 0 (false), and the loop terminates.

Then, the value of sum is printed on the screen.

# C while and do...while Loop

**In this tutorial, you will learn to create while and do...while loop in C programming with the help of examples.**

In programming, loops are used to repeat a block of code until a specified condition is met.

C programming has three types of loops.

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

In the previous tutorial, we learned about for loop. In this tutorial, we will learn about while and do..while loop.

## while loop

The syntax of the while loop is:

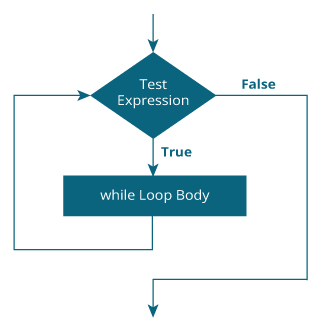
1. while (testExpression)
2. {
3. // statements inside the body of the loop
4. }

### How while loop works?

* The while loop evaluates the test expression inside the parenthesis ().
* If the test expression is true, statements inside the body of while loop are executed. Then, the test expression is evaluated again.
* The process goes on until the test expression is evaluated to false.
* If the test expression is false, the loop terminates (ends).

To learn more about test expression (when the test expression is evaluated to true and false), check out [relational](https://www.programiz.com/c-programming/c-operators#relational) and [logical operators](https://www.programiz.com/c-programming/c-operators#logical).

### Flowchart of while loop



### Example 1: while loop

1. // Print numbers from 1 to 5
2. #include <stdio.h>
3. int main()
4. {
5. int i = 1;
7. while (i <= 5)
8. {
9. printf("%d\n", i);
10. ++i;
11. }
12. return 0;
13. }

**Output**

1

2

3

4

5

Here, we have initialized i to 1.

1. When i is 1, the test expression i <= 5 is true. Hence, the body of the while loop is executed. This prints 1 on the screen and the value of i is increased to 2.
2. Now, i is 2, the test expression i <= 5 is again true. The body of the while loop is executed again. This prints 2 on the screen and the value of i is increased to 3.
3. This process goes on until i becomes 6. When i is 6, the test expression i <= 5 will be false and the loop terminates.

## do...while loop

The do..while loop is similar to the while loop with one important difference. The body of do...while loop is executed at least once. Only then, the test expression is evaluated.

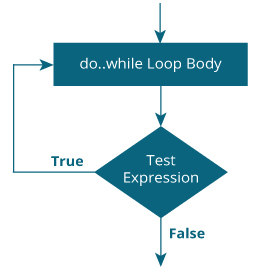
The syntax of the do...while loop is:

1. do
2. {
3. // statements inside the body of the loop
4. }
5. while (testExpression);

### How do...while loop works?

* The body of do...while loop is executed once. Only then, the test expression is evaluated.
* If the test expression is true, the body of the loop is executed again and the test expression is evaluated.
* This process goes on until the test expression becomes false.
* If the test expression is false, the loop ends.

### Flowchart of do...while Loop



### Example 2: do...while loop

1. // Program to add numbers until the user enters zero
2. #include <stdio.h>
3. int main()
4. {
5. double number, sum = 0;
6. // the body of the loop is executed at least once
7. do
8. {
9. printf("Enter a number: ");
10. scanf("%lf", &number);
11. sum += number;
12. }
13. while(number != 0.0);
14. printf("Sum = %.2lf",sum);
15. return 0;
16. }

**Output**

Enter a number: 1.5

Enter a number: 2.4

Enter a number: -3.4

Enter a number: 4.2

Enter a number: 0

Sum = 4.70

# C switch Statement

**In this tutorial, you will learn to create the switch statement in C programming with the help of an example.**

The switch statement allows us to execute one code block among many alternatives.

You can do the same thing with the if...else..if ladder. However, the syntax of the switch statement is much easier to read and write.

## Syntax of switch...case

1. switch (expression)
2. ​{
3. case constant1:
4. // statements
5. break;
6. case constant2:
7. // statements
8. break;
9. .
10. .
11. .
12. default:
13. // default statements
14. }

**How does the switch statement work?**

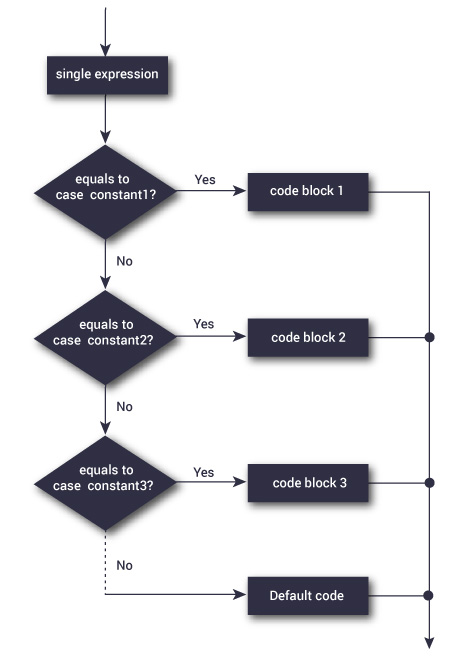
The expression is evaluated once and compared with the values of each case label.

* If there is a match, the corresponding statements after the matching label are executed. For example, if the value of the expression is equal to constant2, statements after case constant2: are executed until break is encountered.
* If there is no match, the default statements are executed.

If we do not use break, all statements after the matching label are executed.

The default clause inside the switch statement is optional.

### switch Statement Flowchart



### Example: Simple Calculator

1. // Program to create a simple calculator
2. #include <stdio.h>
3. int main() {
4. char operator;
5. double n1, n2;
6. printf("Enter an operator (+, -, \*, /): ");
7. scanf("%c", &operator);
8. printf("Enter two operands: ");
9. scanf("%lf %lf",&n1, &n2);
10. switch(operator)
11. {
12. case '+':
13. printf("%.1lf + %.1lf = %.1lf",n1, n2, n1+n2);
14. break;
15. case '-':
16. printf("%.1lf - %.1lf = %.1lf",n1, n2, n1-n2);
17. break;
18. case '\*':
19. printf("%.1lf \* %.1lf = %.1lf",n1, n2, n1\*n2);
20. break;
21. case '/':
22. printf("%.1lf / %.1lf = %.1lf",n1, n2, n1/n2);
23. break;
24. // operator doesn't match any case constant +, -, \*, /
25. default:
26. printf("Error! operator is not correct");
27. }
28. return 0;
29. }

**Output**

Enter an operator (+, -, \*,): -

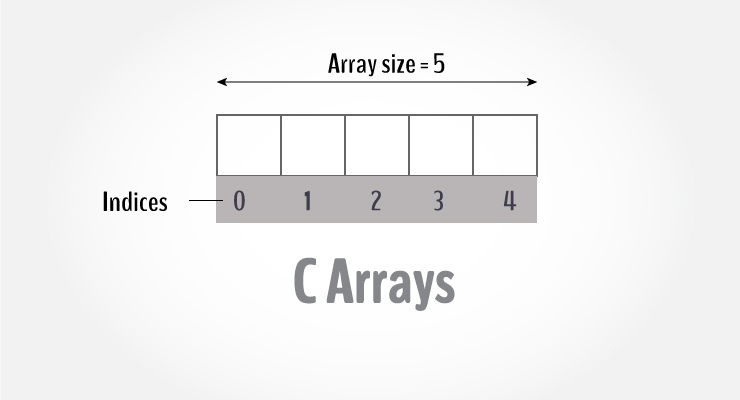
Enter two operands: 32.5

12.4

32.5 - 12.4 = 20.1

# C Arrays

**In this tutorial, you will learn to work with arrays. You will learn to declare, initialize and access array elements of an array with the help of examples.**



An array is a variable that can store multiple values. For example, if you want to store 100 integers, you can create an array for it.

1. int data[100];

## How to declare an array?

dataType arrayName[arraySize];

**For example,**

float mark[5];

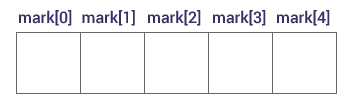
Here, we declared an array, mark, of floating-point type. And its size is 5. Meaning, it can hold 5 floating-point values.

It's important to note that the size and type of an array cannot be changed once it is declared.

## Access Array Elements

You can access elements of an array by indices.

Suppose you declared an array mark as above. The first element is mark[0], the second element is mark[1] and so on.



#### ****Few keynotes****:

* Arrays have 0 as the first index, not 1. In this example, mark[0] is the first element.
* If the size of an array is n, to access the last element, the n-1 index is used. In this example, mark[4]
* Suppose the starting address of mark[0] is **2120d**. Then, the address of the mark[1] will be **2124d**. Similarly, the address of mark[2] will be **2128d** and so on.  
  This is because the size of a float is 4 bytes.

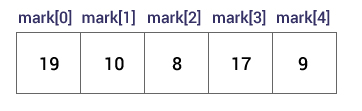
## How to initialize an array?

It is possible to initialize an array during declaration. For example,

1. int mark[5] = {19, 10, 8, 17, 9};

You can also initialize an array like this.

1. int mark[] = {19, 10, 8, 17, 9};

Here, we haven't specified the size. However, the compiler knows its size is 5 as we are initializing it with 5 elements.  


Here,

mark[0] is equal to 19

mark[1] is equal to 10

mark[2] is equal to 8

mark[3] is equal to 17

mark[4] is equal to 9

## Change Value of Array elements

1. int mark[5] = {19, 10, 8, 17, 9}
2. // make the value of the third element to -1
3. mark[2] = -1;
4. // make the value of the fifth element to 0
5. mark[4] = 0;

## Input and Output Array Elements

Here's how you can take input from the user and store it in an array element.

1. // take input and store it in the 3rd element
2. ​scanf("%d", &mark[2]);
3. // take input and store it in the ith element
4. scanf("%d", &mark[i-1]);

Here's how you can print an individual element of an array.

1. // print the first element of the array
2. printf("%d", mark[0]);
3. // print the third element of the array
4. printf("%d", mark[2]);
5. // print ith element of the array
6. printf("%d", mark[i-1]);

## Example 1: Array Input/Output

1. // Program to take 5 values from the user and store them in an array
2. // Print the elements stored in the array
3. #include <stdio.h>
4. int main() {
5. int values[5];
6. printf("Enter 5 integers: ");
7. // taking input and storing it in an array
8. for(int i = 0; i < 5; ++i) {
9. scanf("%d", &values[i]);
10. }
11. printf("Displaying integers: ");
12. // printing elements of an array
13. for(int i = 0; i < 5; ++i) {
14. printf("%d\n", values[i]);
15. }
16. return 0;
17. }

**Output**

Enter 5 integers: 1

-3

34

0

3

Displaying integers: 1

-3

34

0

3

Here, we have used a for loop to take 5 inputs from the user and store them in an array. Then, using another for loop, these elements are displayed on the screen.

## Example 2: Calculate Average

1. // Program to find the average of n numbers using arrays
2. #include <stdio.h>
3. int main()
4. {
5. int marks[10], i, n, sum = 0, average;
6. printf("Enter number of elements: ");
7. scanf("%d", &n);
8. for(i=0; i<n; ++i)
9. {
10. printf("Enter number%d: ",i+1);
11. scanf("%d", &marks[i]);
13. // adding integers entered by the user to the sum variable
14. sum += marks[i];
15. }
16. average = sum/n;
17. printf("Average = %d", average);
18. return 0;
19. }

**Output**

Enter n: 5

Enter number1: 45

Enter number2: 35

Enter number3: 38

Enter number4: 31

Enter number5: 49

Average = 39

Here, we have computed the average of n numbers entered by the user.

### Access elements out of its bound!

Suppose you declared an array of 10 elements. Let's say,

int testArray[10];

You can access the array elements from testArray[0] to testArray[9].

Now let's say if you try to access testArray[12]. The element is not available. This may cause unexpected output (undefined behavior). Sometimes you might get an error and some other time your program may run correctly.

Hence, you should never access elements of an array outside of its bound.

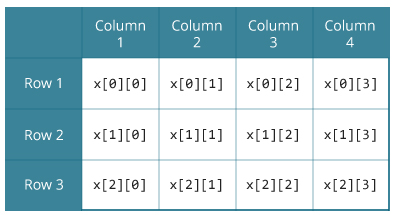
# C Multidimensional Arrays

**In this tutorial, you will learn to work with multidimensional arrays (two-dimensional and three-dimensional arrays) with the help of examples.**

In C programming, you can create an array of arrays. These arrays are known as multidimensional arrays. For example,

1. float x[3][4];

Here, x is a two-dimensional (2d) array. The array can hold 12 elements. You can think the array as a table with 3 rows and each row has 4 columns.



Similarly, you can declare a three-dimensional (3d) array. For example,

1. float y[2][4][3];

Here, the array y can hold 24 elements.

## Initializing a multidimensional array

Here is how you can initialize two-dimensional and three-dimensional arrays:

### Initialization of a 2d array

1. // Different ways to initialize two-dimensional array
2. int c[2][3] = {{1, 3, 0}, {-1, 5, 9}};
4. int c[][3] = {{1, 3, 0}, {-1, 5, 9}};
6. int c[2][3] = {1, 3, 0, -1, 5, 9};

### Initialization of a 3d array

You can initialize a three-dimensional array in a similar way like a two-dimensional array. Here's an example,

1. int test[2][3][4] = {
2. {{3, 4, 2, 3}, {0, -3, 9, 11}, {23, 12, 23, 2}},
3. {{13, 4, 56, 3}, {5, 9, 3, 5}, {3, 1, 4, 9}}};

### Example 1: Two-dimensional array to store and print values

1. // C program to store temperature of two cities of a week and display it.
2. #include <stdio.h>
3. const int CITY = 2;
4. const int WEEK = 7;
5. int main()
6. {
7. int temperature[CITY][WEEK];
8. // Using nested loop to store values in a 2d array
9. for (int i = 0; i < CITY; ++i)
10. {
11. for (int j = 0; j < WEEK; ++j)
12. {
13. printf("City %d, Day %d: ", i + 1, j + 1);
14. scanf("%d", &temperature[i][j]);
15. }
16. }
17. printf("\nDisplaying values: \n\n");
18. // Using nested loop to display vlues of a 2d array
19. for (int i = 0; i < CITY; ++i)
20. {
21. for (int j = 0; j < WEEK; ++j)
22. {
23. printf("City %d, Day %d = %d\n", i + 1, j + 1, temperature[i][j]);
24. }
25. }
26. return 0;
27. }

**Output**

City 1, Day 1: 33

City 1, Day 2: 34

City 1, Day 3: 35

City 1, Day 4: 33

City 1, Day 5: 32

City 1, Day 6: 31

City 1, Day 7: 30

City 2, Day 1: 23

City 2, Day 2: 22

City 2, Day 3: 21

City 2, Day 4: 24

City 2, Day 5: 22

City 2, Day 6: 25

City 2, Day 7: 26

Displaying values:

City 1, Day 1 = 33

City 1, Day 2 = 34

City 1, Day 3 = 35

City 1, Day 4 = 33

City 1, Day 5 = 32

City 1, Day 6 = 31

City 1, Day 7 = 30

City 2, Day 1 = 23

City 2, Day 2 = 22

City 2, Day 3 = 21

City 2, Day 4 = 24

City 2, Day 5 = 22

City 2, Day 6 = 25

City 2, Day 7 = 26

### Example 2: Sum of two matrices

1. // C program to find the sum of two matrices of order 2\*2
2. #include <stdio.h>
3. int main()
4. {
5. float a[2][2], b[2][2], result[2][2];
6. // Taking input using nested for loop
7. printf("Enter elements of 1st matrix\n");
8. for (int i = 0; i < 2; ++i)
9. for (int j = 0; j < 2; ++j)
10. {
11. printf("Enter a%d%d: ", i + 1, j + 1);
12. scanf("%f", &a[i][j]);
13. }
14. // Taking input using nested for loop
15. printf("Enter elements of 2nd matrix\n");
16. for (int i = 0; i < 2; ++i)
17. for (int j = 0; j < 2; ++j)
18. {
19. printf("Enter b%d%d: ", i + 1, j + 1);
20. scanf("%f", &b[i][j]);
21. }
22. // adding corresponding elements of two arrays
23. for (int i = 0; i < 2; ++i)
24. for (int j = 0; j < 2; ++j)
25. {
26. result[i][j] = a[i][j] + b[i][j];
27. }
28. // Displaying the sum
29. printf("\nSum Of Matrix:");
30. for (int i = 0; i < 2; ++i)
31. for (int j = 0; j < 2; ++j)
32. {
33. printf("%.1f\t", result[i][j]);
34. if (j == 1)
35. printf("\n");
36. }
37. return 0;
38. }

**Output**

Enter elements of 1st matrix

Enter a11: 2;

Enter a12: 0.5;

Enter a21: -1.1;

Enter a22: 2;

Enter elements of 2nd matrix

Enter b11: 0.2;

Enter b12: 0;

Enter b21: 0.23;

Enter b22: 23;

Sum Of Matrix:

2.2 0.5

-0.9 25.0

### Example 3: Three-dimensional array

1. // C Program to store and print 12 values entered by the user
2. #include <stdio.h>
3. int main()
4. {
5. int test[2][3][2];
6. printf("Enter 12 values: \n");
7. for (int i = 0; i < 2; ++i)
8. {
9. for (int j = 0; j < 3; ++j)
10. {
11. for (int k = 0; k < 2; ++k)
12. {
13. scanf("%d", &test[i][j][k]);
14. }
15. }
16. }
17. // Printing values with proper index.
18. printf("\nDisplaying values:\n");
19. for (int i = 0; i < 2; ++i)
20. {
21. for (int j = 0; j < 3; ++j)
22. {
23. for (int k = 0; k < 2; ++k)
24. {
25. printf("test[%d][%d][%d] = %d\n", i, j, k, test[i][j][k]);
26. }
27. }
28. }
29. return 0;
30. }

**Output**

Enter 12 values:

1

2

3

4

5

6

7

8

9

10

11

12

Displaying Values:

test[0][0][0] = 1

test[0][0][1] = 2

test[0][1][0] = 3

test[0][1][1] = 4

test[0][2][0] = 5

test[0][2][1] = 6

test[1][0][0] = 7

test[1][0][1] = 8

test[1][1][0] = 9

test[1][1][1] = 10

test[1][2][0] = 11

test[1][2][1] = 12

# C Functions

**In this tutorial, you will be introduced to functions (both user-defined and standard library functions) in C programming. Also, you will learn why functions are used in programming.**

A function is a block of code that performs a specific task.

Suppose, you need to create a program to create a circle and color it. You can create two functions to solve this problem:

* create a circle function
* create a color function

Dividing a complex problem into smaller chunks makes our program easy to understand to reuse.

## Types of function

There are two types of function in C programming:

* [Standard library functions](https://www.programiz.com/c-programming/library-function)
* [User-defined functions](https://www.programiz.com/c-programming/c-user-defined-functions)

### Standard library functions

The standard library functions are built-in functions in C programming.

These functions are defined in header files. For example,

* The printf() is a standard library function to send formatted output to the screen (display output on the screen). This function is defined in the stdio.h header file.  
  Hence, to use the printf() function, we need to include the stdio.h header file using #include <stdio.h>.
* The sqrt() function calculates the square root of a number. The function is defined in the math.h header file.

Visit [standard library functions in C programming](https://www.programiz.com/c-programming/library-function) to learn more.

### User-defined function

You can also create functions as per your need. Such functions created by the user are known as user-defined functions.

## How user-defined function works?

#include <stdio.h>

void functionName()

{

... .. ...

... .. ...

}

int main()

{

... .. ...

... .. ...

functionName();

... .. ...

... .. ...

}

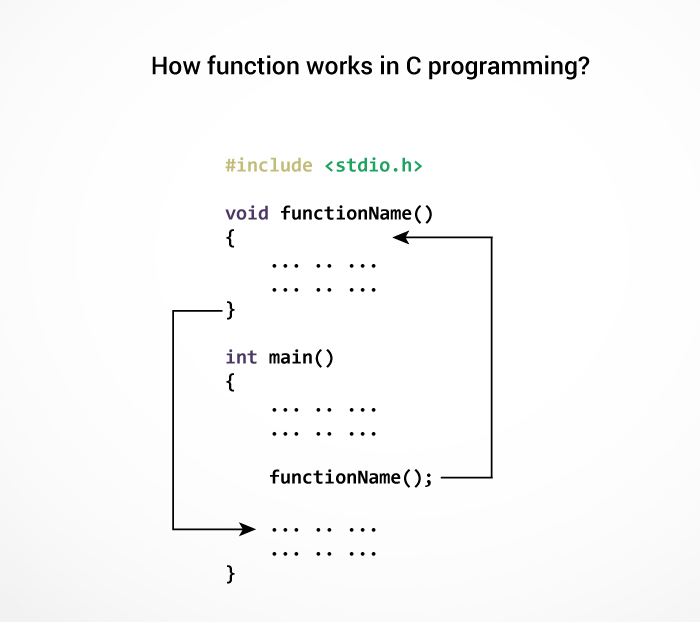
The execution of a C program begins from the main() function.

When the compiler encounters functionName();, control of the program jumps to

void functionName()

And, the compiler starts executing the codes inside functionName().

The control of the program jumps back to the main() function once code inside the function definition is executed.



Note, function names are identifiers and should be unique.

This is just an overview of user-defined functions. Visit these pages to learn more on:

* [User-defined Function in C programming](https://www.programiz.com/c-programming/c-user-defined-functions)
* [Types of user-defined Functions](https://www.programiz.com/c-programming/types-user-defined-functions)

### Advantages of user-defined function

1. The program will be easier to understand, maintain and debug.
2. Reusable codes that can be used in other programs
3. A large program can be divided into smaller modules. Hence, a large project can be divided among many programmers.

# C User-defined functions

**In this tutorial, you will learn to create user-defined functions in C programming with the help of an example.**

A function is a block of code that performs a specific task.

C allows you to define functions according to your need. These functions are known as user-defined functions. For example:

Suppose, you need to create a circle and color it depending upon the radius and color. You can create two functions to solve this problem:

* createCircle() function
* color() function

## Example: User-defined function

Here is an example to add two integers. To perform this task, we have created an user-defined addNumbers().

1. #include <stdio.h>
2. int addNumbers(int a, int b); // function prototype
3. int main()
4. {
5. int n1,n2,sum;
6. printf("Enters two numbers: ");
7. scanf("%d %d",&n1,&n2);
8. sum = addNumbers(n1, n2); // function call
9. printf("sum = %d",sum);
10. return 0;
11. }
12. int addNumbers(int a, int b) // function definition
13. {
14. int result;
15. result = a+b;
16. return result; // return statement
17. }

## Function prototype

A function prototype is simply the declaration of a function that specifies function's name, parameters and return type. It doesn't contain function body.

A function prototype gives information to the compiler that the function may later be used in the program.

### Syntax of function prototype

returnType functionName(type1 argument1, type2 argument2, ...);

In the above example, int addNumbers(int a, int b); is the function prototype which provides the following information to the compiler:

1. name of the function is addNumbers()
2. return type of the function is int
3. two arguments of type int are passed to the function

The function prototype is not needed if the user-defined function is defined before the main() function.

## Calling a function

Control of the program is transferred to the user-defined function by calling it.

### Syntax of function call

functionName(argument1, argument2, ...);

In the above example, the function call is made using addNumbers(n1, n2); statement inside the main() function.

## Function definition

Function definition contains the block of code to perform a specific task. In our example, adding two numbers and returning it.

#### Syntax of function definition

returnType functionName(type1 argument1, type2 argument2, ...)

{

//body of the function

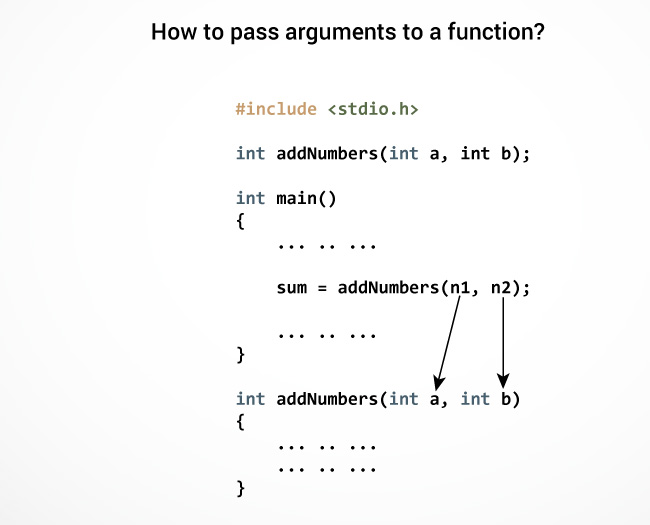
}

When a function is called, the control of the program is transferred to the function definition. And, the compiler starts executing the codes inside the body of a function.

## Passing arguments to a function

In programming, argument refers to the variable passed to the function. In the above example, two variables n1 and n2 are passed during the function call.

The parameters a and b accepts the passed arguments in the function definition. These arguments are called formal parameters of the function.



The type of arguments passed to a function and the formal parameters must match, otherwise, the compiler will throw an error.

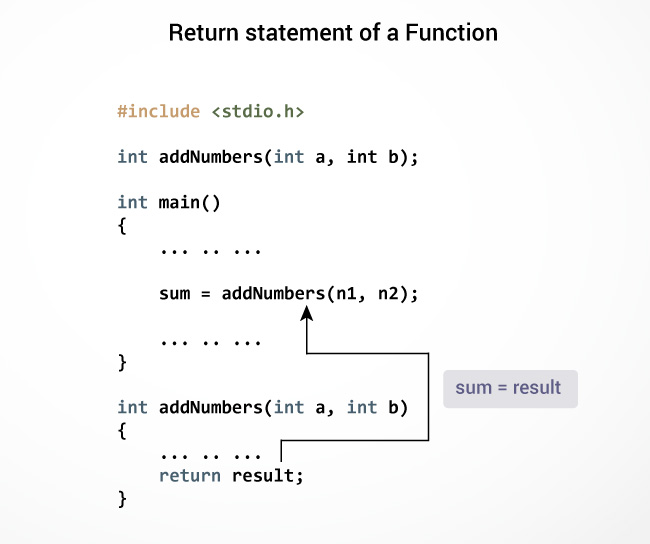
If n1 is of char type, a also should be of char type. If n2 is of float type, variable b also should be of float type.

A function can also be called without passing an argument.

## Return Statement

The return statement terminates the execution of a function and returns a value to the calling function. The program control is transferred to the calling function after the return statement.

In the above example, the value of the result variable is returned to the main function. The sum variable in the main() function is assigned this value.



### Syntax of return statement

return (expression);

For example,

return a;

return (a+b);

The type of value returned from the function and the return type specified in the function prototype and function definition must match.

# Types of User-defined Functions in C Programming

**In this tutorial, you will learn about different approaches you can take to solve the same problem using functions.**

These 4 programs below check whether the integer entered by the user is a prime number or not.

The output of all these programs below is the same, and we have created a user-defined function in each example. However, the approach we have taken in each example is different.

## Example 1: No arguments passed and no return value

1. #include <stdio.h>
2. void checkPrimeNumber();
3. int main()
4. {
5. checkPrimeNumber(); // argument is not passed
6. return 0;
7. }
8. // return type is void meaning doesn't return any value
9. void checkPrimeNumber()
10. {
11. int n, i, flag = 0;
12. printf("Enter a positive integer: ");
13. scanf("%d",&n);
14. for(i=2; i <= n/2; ++i)
15. {
16. if(n%i == 0)
17. {
18. flag = 1;
19. }
20. }
21. if (flag == 1)
22. printf("%d is not a prime number.", n);
23. else
24. printf("%d is a prime number.", n);
25. }

The checkPrimeNumber() function takes input from the user, checks whether it is a prime number or not and displays it on the screen.

The empty parentheses in checkPrimeNumber(); statement inside the main() function indicates that no argument is passed to the function.

The return type of the function is void. Hence, no value is returned from the function.

## Example 2: No arguments passed but a return value

1. #include <stdio.h>
2. int getInteger();
3. int main()
4. {
5. int n, i, flag = 0;
6. // no argument is passed
7. n = getInteger();
8. for(i=2; i<=n/2; ++i)
9. {
10. if(n%i==0){
11. flag = 1;
12. break;
13. }
14. }
15. if (flag == 1)
16. printf("%d is not a prime number.", n);
17. else
18. printf("%d is a prime number.", n);
19. return 0;
20. }
21. // returns integer entered by the user
22. int getInteger()
23. {
24. int n;
25. printf("Enter a positive integer: ");
26. scanf("%d",&n);
27. return n;
28. }

The empty parentheses in the n = getInteger(); statement indicates that no argument is passed to the function. And, the value returned from the function is assigned to n.

Here, the getInteger() function takes input from the user and returns it. The code to check whether a number is prime or not is inside the main() function.

## Example 3: Argument passed but no return value

1. #include <stdio.h>
2. void checkPrimeAndDisplay(int n);
3. int main()
4. {
5. int n;
6. printf("Enter a positive integer: ");
7. scanf("%d",&n);
8. // n is passed to the function
9. checkPrimeAndDisplay(n);
10. return 0;
11. }
12. // return type is void meaning doesn't return any value
13. void checkPrimeAndDisplay(int n)
14. {
15. int i, flag = 0;
16. for(i=2; i <= n/2; ++i)
17. {
18. if(n%i == 0){
19. flag = 1;
20. break;
21. }
22. }
23. if(flag == 1)
24. printf("%d is not a prime number.",n);
25. else
26. printf("%d is a prime number.", n);
27. }

The integer value entered by the user is passed to the checkPrimeAndDisplay() function.

Here, the checkPrimeAndDisplay() function checks whether the argument passed is a prime number or not and displays the appropriate message.

## Example 4: Argument passed and a return value

1. #include <stdio.h>
2. int checkPrimeNumber(int n);
3. int main()
4. {
5. int n, flag;
6. printf("Enter a positive integer: ");
7. scanf("%d",&n);
8. // n is passed to the checkPrimeNumber() function
9. // the returned value is assigned to the flag variable
10. flag = checkPrimeNumber(n);
11. if(flag == 1)
12. printf("%d is not a prime number",n);
13. else
14. printf("%d is a prime number",n);
15. return 0;
16. }
17. // int is returned from the function
18. int checkPrimeNumber(int n)
19. {
20. int i;
21. for(i=2; i <= n/2; ++i)
22. {
23. if(n%i == 0)
24. return 1;
25. }
26. return 0;
27. }

The input from the user is passed to the checkPrimeNumber() function.

The checkPrimeNumber() function checks whether the passed argument is prime or not.

If the passed argument is a prime number, the function returns 0. If the passed argument is a non-prime number, the function returns 1. The return value is assigned to the flag variable.

Depending on whether flag is 0 or 1, an appropriate message is printed from the main() function.

## Which approach is better?

Well, it depends on the problem you are trying to solve. In this case, passing argument and returning a value from the function (example 4) is better.

A function should perform a specific task. The checkPrimeNumber() function doesn't take input from the user nor it displays the appropriate message. It only checks whether a number is prime or not.

# C Recursion

**In this tutorial, you will learn to write recursive functions in C programming with the help of an example.**

A function that calls itself is known as a recursive function. And, this technique is known as recursion.

### How recursion works?

void recurse()

{

... .. ...

recurse();

... .. ...

}

int main()

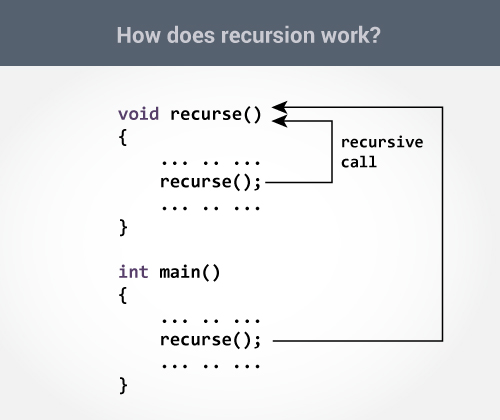
{

... .. ...

recurse();

... .. ...

}



The recursion continues until some condition is met to prevent it.

To prevent infinite recursion, [if...else statement](https://www.programiz.com/c-programming/c-if-else-statement) (or similar approach) can be used where one branch makes the recursive call, and other doesn't.

### Example: Sum of Natural Numbers Using Recursion

1. #include <stdio.h>
2. int sum(int n);
3. int main() {
4. int number, result;
5. printf("Enter a positive integer: ");
6. scanf("%d", &number);
7. result = sum(number);
8. printf("sum = %d", result);
9. return 0;
10. }
11. int sum(int n) {
12. if (n != 0)
13. // sum() function calls itself
14. return n + sum(n-1);
15. else
16. return n;
17. }

**Output**

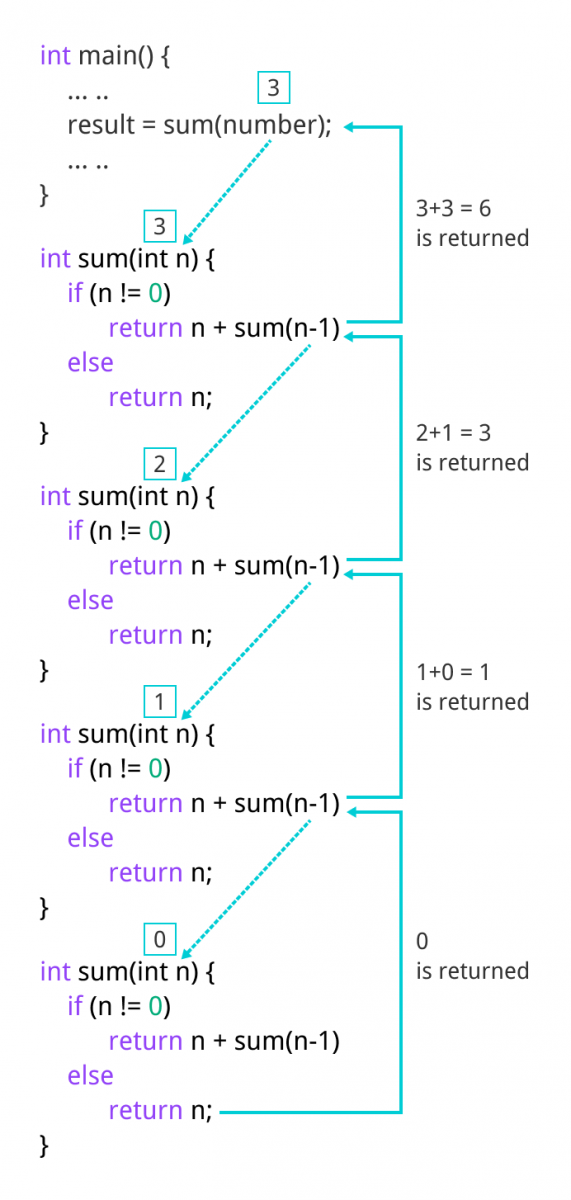
Enter a positive integer:3

sum = 6

Initially, the sum() is called from the main() function with number passed as an argument.

Suppose, the value of n inside sum() is 3 initially. During the next function call, 2 is passed to the sum() function. This process continues until n is equal to 0.

When n is equal to 0, the if condition fails and the else part is executed returning the sum of integers ultimately to the main() function.



### Advantages and Disadvantages of Recursion

Recursion makes program elegant. However, if performance is vital, use loops instead as recursion is usually much slower.

That being said, recursion is an important concept. It is frequently used in [data structure and algorithms](https://www.programiz.com/dsa). For example, it is common to use recursion in problems such as tree traversal.

# C Storage Class

**In this tutorial, you will learn about scope and lifetime of local and global variables. Also, you will learn about static and register variables.**

Every variable in C programming has two properties: type and storage class.

Type refers to the data type of a variable. And, storage class determines the scope, visibility and lifetime of a variable.

There are 4 types of storage class:

1. automatic
2. external
3. static
4. register

## Local Variable

The variables declared inside a block are automatic or local variables. The local variables exist only inside the block in which it is declared.

Let's take an example.

1. #include <stdio.h>
2. int main(void) {
4. for (int i = 0; i < 5; ++i) {
5. printf("C programming");
6. }
8. // Error: i is not declared at this point
9. printf("%d", i);
10. return 0;
11. }

When you run the above program, you will get an error undeclared identifier i. It's because i is declared inside the for loop block. Outside of the block, it's undeclared.

Let's take another example.

1. int main() {
2. int n1; // n1 is a local variable to main()
3. }
4. void func() {
5. int n2; // n2 is a local variable to func()
6. }

In the above example, n1 is local to main() and n2 is local to func().

This means you cannot access the n1 variable inside func() as it only exists inside main(). Similarly, you cannot access the n2 variable inside main() as it only exists inside func().

## Global Variable

Variables that are declared outside of all functions are known as external or global variables. They are accessible from any function inside the program.

### Example 1: Global Variable

1. #include <stdio.h>
2. void display();
3. int n = 5; // global variable
4. int main()
5. {
6. ++n;
7. display();
8. return 0;
9. }
10. void display()
11. {
12. ++n;
13. printf("n = %d", n);
14. }

**Output**

n = 7

Suppose, a global variable is declared in file1. If you try to use that variable in a different file file2, the compiler will complain. To solve this problem, keyword extern is used in file2 to indicate that the external variable is declared in another file.

## Register Variable

The register keyword is used to declare register variables. Register variables were supposed to be faster than local variables.

However, modern compilers are very good at code optimization, and there is a rare chance that using register variables will make your program faster.

Unless you are working on embedded systems where you know how to optimize code for the given application, there is no use of register variables.

## Static Variable

A static variable is declared by using the static keyword. For example;

static int i;

The value of a static variable persists until the end of the program.

### Example 2: Static Variable

1. #include <stdio.h>
2. void display();
3. int main()
4. {
5. display();
6. display();
7. }
8. void display()
9. {
10. static int c = 1;
11. c += 5;
12. printf("%d ",c);
13. }

**Output**

6 11

During the first function call, the value of c is initialized to 1. Its value is increased by 5. Now, the value of c is 6, which is printed on the screen.

During the second function call, c is not initialized to 1 again. It's because c is a static variable. The value c is increased by 5. Now, its value will be 11, which is printed on the screen.

# C struct

**In this tutorial, you'll learn about struct types in C Programming. You will learn to define and use structures with the help of examples.**

In C programming, a struct (or structure) is a collection of variables (can be of different types) under a single name.

## How to define structures?

Before you can create structure variables, you need to define its data type. To define a struct, the struct keyword is used.

### Syntax of struct

struct structureName

{

dataType member1;

dataType member2;

...

};

Here is an example:

struct Person

{

char name[50];

int citNo;

float salary;

};

Here, a derived type struct Person is defined. Now, you can create variables of this type.

## Create struct variables

When a struct type is declared, no storage or memory is allocated. To allocate memory of a given structure type and work with it, we need to create variables.

Here's how we create structure variables:

struct Person

{

char name[50];

int citNo;

float salary;

};

int main()

{

struct Person person1, person2, p[20];

return 0;

}

Another way of creating a struct variable is:

struct Person

{

char name[50];

int citNo;

float salary;

} person1, person2, p[20];

In both cases, two variables person1, person2, and an array variable p having 20 elements of type struct Person are created.

## Access members of a structure

There are two types of operators used for accessing members of a structure.

1. . - Member operator
2. -> - Structure pointer operator (will be discussed in the next tutorial)

Suppose, you want to access the salary of person2. Here's how you can do it.

person2.salary

### Example: Add two distances

1. // Program to add two distances (feet-inch)
2. #include <stdio.h>
3. struct Distance
4. {
5. int feet;
6. float inch;
7. } dist1, dist2, sum;
8. int main()
9. {
10. printf("1st distance\n");
11. printf("Enter feet: ");
12. scanf("%d", &dist1.feet);
13. printf("Enter inch: ");
14. scanf("%f", &dist1.inch);
15. printf("2nd distance\n");
16. printf("Enter feet: ");
17. scanf("%d", &dist2.feet);
18. printf("Enter inch: ");
19. scanf("%f", &dist2.inch);
20. // adding feet
21. sum.feet = dist1.feet + dist2.feet;
22. // adding inches
23. sum.inch = dist1.inch + dist2.inch;
24. // changing to feet if inch is greater than 12
25. while (sum.inch >= 12)
26. {
27. ++sum.feet;
28. sum.inch = sum.inch - 12;
29. }
30. printf("Sum of distances = %d\'-%.1f\"", sum.feet, sum.inch);
31. return 0;
32. }

**Output**

1st distance

Enter feet: 12

Enter inch: 7.9

2nd distance

Enter feet: 2

Enter inch: 9.8

Sum of distances = 15'-5.7"

### Keyword typedef

We use the typedef keyword to create an alias name for data types. It is commonly used with structures to simplify the syntax of declaring variables.

**This code**

struct Distance{

int feet;

float inch;

};

int main() {

structure Distance d1, d2;

}

**is equivalent to**

typedef struct Distance{

int feet;

float inch;

} distances;

int main() {

distances d1, d2;

}

## Nested Structures

You can create structures within a structure in C programming. For example,

struct complex

{

int imag;

float real;

};

struct number

{

struct complex comp;

int integers;

} num1, num2;

Suppose, you want to set imag of num2 variable to 11. Here's how you can do it:

num2.comp.imag = 11;

## Why structs in C?

Suppose, you want to store information about a person: his/her name, citizenship number, and salary. You can create different variables name, citNo and salary to store this information.

What if you need to store information of more than one person? Now, you need to create different variables for each information per person: name1, citNo1, salary1, name2, citNo2, salary2, etc.

A better approach would be to have a collection of all related information under a single name Person structure and use it for every person.

# C structs and Pointers

**In this tutorial, you'll learn to use pointers to access members of structs in C programming. You will also learn to dynamically allocate memory of struct types.**

Before you learn about how pointers can be used with structs, be sure to check these tutorials:

* [C Pointers](https://www.programiz.com/c-programming/c-pointers)
* [C struct](https://www.programiz.com/c-programming/c-structures)

## C Pointers to struct

Here's how you can create pointers to structs.

1. struct name {
2. member1;
3. member2;
4. .
5. .
6. };
7. int main()
8. {
9. struct name \*ptr, Harry;
10. }

Here, ptr is a pointer to struct.

## Example: Access members using Pointer

To access members of a structure using pointers, we use the -> operator.

1. #include <stdio.h>
2. struct person
3. {
4. int age;
5. float weight;
6. };
7. int main()
8. {
9. struct person \*personPtr, person1;
10. personPtr = &person1;
11. printf("Enter age: ");
12. scanf("%d", &personPtr->age);
13. printf("Enter weight: ");
14. scanf("%f", &personPtr->weight);
15. printf("Displaying:\n");
16. printf("Age: %d\n", personPtr->age);
17. printf("weight: %f", personPtr->weight);
18. return 0;
19. }

In this example, the address of person1 is stored in the personPtr pointer using personPtr = &person1;.

Now, you can access the members of person1 using the personPtr pointer.

By the way,

* personPtr->age is equivalent to (\*personPtr).age
* personPtr->weight is equivalent to (\*personPtr).weight

## Dynamic memory allocation of structs

Before you proceed this section, we recommend you to check [C dynamic memory allocation](https://www.programiz.com/c-programming/c-dynamic-memory-allocation).

Sometimes, the number of struct variables you declared may be insufficient. You may need to allocate memory during run-time. Here's how you can achieve this in C programming.

### Example: Dynamic memory allocation of structs

1. #include <stdio.h>
2. #include <stdlib.h>
3. struct person {
4. int age;
5. float weight;
6. char name[30];
7. };
8. int main()
9. {
10. struct person \*ptr;
11. int i, n;
12. printf("Enter the number of persons: ");
13. scanf("%d", &n);
14. // allocating memory for n numbers of struct person
15. ptr = (struct person\*) malloc(n \* sizeof(struct person));
16. for(i = 0; i < n; ++i)
17. {
18. printf("Enter first name and age respectively: ");
19. // To access members of 1st struct person,
20. // ptr->name and ptr->age is used
21. // To access members of 2nd struct person,
22. // (ptr+1)->name and (ptr+1)->age is used
23. scanf("%s %d", (ptr+i)->name, &(ptr+i)->age);
24. }
25. printf("Displaying Information:\n");
26. for(i = 0; i < n; ++i)
27. printf("Name: %s\tAge: %d\n", (ptr+i)->name, (ptr+i)->age);
28. return 0;
29. }

When you run the program, the output will be:

Enter the number of persons: 2

Enter first name and age respectively: Harry 24

Enter first name and age respectively: Gary 32

Displaying Information:

Name: Harry Age: 24

Name: Gary Age: 32

In the above example, n number of struct variables are created where n is entered by the user.

To allocate the memory for n number of struct person, we used,

1. ptr = (struct person\*) malloc(n \* sizeof(struct person));

Then, we used the ptr pointer to access elements of person.

# C Structure and Function

**In this tutorial, you'll learn to pass struct variables as arguments to a function. You will learn to return struct from a function with the help of examples.**

Similar to variables of built-in types, you can also pass structure variables to a function.

## Passing structs to functions

We recommended you to learn these tutorials before you learn how to pass structs to functions.

* [C structures](https://www.programiz.com/c-programming/c-structures)
* [C functions](https://www.programiz.com/c-programming/c-functions)
* [User-defined Function](https://www.programiz.com/c-programming/c-user-defined-functions)

Here's how you can pass structures to a function

1. #include <stdio.h>
2. struct student
3. {
4. char name[50];
5. int age;
6. };
7. // function prototype
8. void display(struct student s);
9. int main()
10. {
11. struct student s1;
12. printf("Enter name: ");
13. scanf("%[^\n]%\*c", s1.name);
14. printf("Enter age: ");
15. scanf("%d", &s1.age);
17. display(s1); // passing struct as an argument
19. return 0;
20. }
21. void display(struct student s)
22. {
23. printf("\nDisplaying information\n");
24. printf("Name: %s", s.name);
25. printf("\nAge: %d", s.age);
26. }

**Output**

Enter name: Bond

Enter age: 13

Displaying information

Name: Bond

Age: 13

Here, a struct variable s1 of type struct student is created. The variable is passed to the display() function using display(s1); statement.

## Return struct from a function

Here's how you can return structure from a function:

1. #include <stdio.h>
2. struct student
3. {
4. char name[50];
5. int age;
6. };
7. // function prototype
8. struct student getInformation();
9. int main()
10. {
11. struct student s;
12. s = getInformation();
13. printf("\nDisplaying information\n");
14. printf("Name: %s", s.name);
15. printf("\nRoll: %d", s.age);
17. return 0;
18. }
19. struct student getInformation()
20. {
21. struct student s1;
22. printf("Enter name: ");
23. scanf ("%[^\n]%\*c", s1.name);
24. printf("Enter age: ");
25. scanf("%d", &s1.age);
27. return s1;
28. }

Here, the getInformation() function is called using s = getInformation(); statement. The function returns a structure of type struct student. The returned structure is displayed from the main() function.

Notice that, the return type of getInformation() is also struct student.

## Passing struct by reference

You can also pass structs by reference (in a similar way like you pass variables of built-in type by reference). We suggest you to read [pass by reference](https://www.programiz.com/c-programming/c-pointer-functions) tutorial before you proceed.

During pass by reference, the memory addresses of struct variables are passed to the function.

1. #include <stdio.h>
2. typedef struct Complex
3. {
4. float real;
5. float imag;
6. } complex;
7. void addNumbers(complex c1, complex c2, complex \*result);
8. int main()
9. {
10. complex c1, c2, result;
11. printf("For first number,\n");
12. printf("Enter real part: ");
13. scanf("%f", &c1.real);
14. printf("Enter imaginary part: ");
15. scanf("%f", &c1.imag);
16. printf("For second number, \n");
17. printf("Enter real part: ");
18. scanf("%f", &c2.real);
19. printf("Enter imaginary part: ");
20. scanf("%f", &c2.imag);
21. addNumbers(c1, c2, &result);
22. printf("\nresult.real = %.1f\n", result.real);
23. printf("result.imag = %.1f", result.imag);
25. return 0;
26. }
27. void addNumbers(complex c1, complex c2, complex \*result)
28. {
29. result->real = c1.real + c2.real;
30. result->imag = c1.imag + c2.imag;
31. }

**Output**

For first number,

Enter real part: 1.1

Enter imaginary part: -2.4

For second number,

Enter real part: 3.4

Enter imaginary part: -3.2

result.real = 4.5

result.imag = -5.6

In the above program, three structure variables c1, c2 and the address of result is passed to the addNumbers() function. Here, result is passed by reference.

When the result variable inside the addNumbers() is altered, the result variable inside the main() function is also altered accordingly.

# C Unions

**In this tutorial, you'll learn about unions in C programming. More specifically, how to create unions, access its members and learn the differences between unions and structures.**

A union is a user-defined type similar to structs in C programming. We recommend you to learn [C structs](https://www.programiz.com/c-programming/c-structures) before you check this tutorial.

## How to define a union?

We use the union keyword to define unions. Here's an example:

1. union car
2. {
3. char name[50];
4. int price;
5. };

The above code defines a derived type union car.

## Create union variables

When a union is defined, it creates a user-defined type. However, no memory is allocated. To allocate memory for a given union type and work with it, we need to create variables.

Here's how we create union variables.

1. union car
2. {
3. char name[50];
4. int price;
5. };
6. int main()
7. {
8. union car car1, car2, \*car3;
9. return 0;
10. }

Another way of creating union variables is:

1. union car
2. {
3. char name[50];
4. int price;
5. } car1, car2, \*car3;

In both cases, union variables car1, car2, and a union pointer car3 of union car type are created.

### Access members of a union

We use the . operator to access members of a union. To access pointer variables, we use also use the -> operator.

In the above example,

* To access price for car1, car1.price is used.
* To access price using car3, either (\*car3).price or car3->price can be used.

## Difference between unions and structures

Let's take an example to demonstrate the difference between unions and structures:

1. #include <stdio.h>
2. union unionJob
3. {
4. //defining a union
5. char name[32];
6. float salary;
7. int workerNo;
8. } uJob;
9. struct structJob
10. {
11. char name[32];
12. float salary;
13. int workerNo;
14. } sJob;
15. int main()
16. {
17. printf("size of union = %d bytes", sizeof(uJob));
18. printf("\nsize of structure = %d bytes", sizeof(sJob));
19. return 0;
20. }

**Output**

size of union = 32

size of structure = 40

**Why this difference in the size of union and structure variables?**

Here, the size of sJob is 40 bytes because

* the size of name[32] is 32 bytes
* the size of salary is 4 bytes
* the size of workerNo is 4 bytes

However, the size of uJob is 32 bytes. It's because the size of a union variable will always be the size of its largest element. In the above example, the size of its largest element, (name[32]), is 32 bytes.

### Only one union member can be accessed at a time

You can access all members of a structure at once as sufficient memory is allocated for all members. However, it's not the case in unions. You can only access a single member of a union at one time. Let's see an example.

1. #include <stdio.h>
2. union Job
3. {
4. float salary;
5. int workerNo;
6. } j;
7. int main()
8. {
9. j.salary = 12.3;
10. j.workerNo = 100;
11. printf("Salary = %.1f\n", j.salary);
12. printf("Number of workers = %d", j.workerNo);
13. return 0;
14. }

**Output**

Salary = 0.0

Number of workers = 100

# C Pointers

**In this tutorial, you'll learn about pointers; what pointers are, how do you use them and the common mistakes you might face when working with them with the help of examples.**

Pointers are powerful features of C and C++ programming. Before we learn pointers, let's learn about addresses in C programming.

## Address in C

If you have a variable var in your program, &var will give you its address in the memory.

We have used address numerous times while using the scanf() function.

1. scanf("%d", &var);

Here, the value entered by the user is stored in the address of var variable. Let's take a working example.

1. #include <stdio.h>
2. int main()
3. {
4. int var = 5;
5. printf("var: %d\n", var);
6. // Notice the use of & before var
7. printf("address of var: %p", &var);
8. return 0;
9. }

**Output**

var: 5

address of var: 2686778

**Note:** You will probably get a different address when you run the above code.

## C Pointers

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

### Pointer Syntax

Here is how we can declare pointers.

1. int\* p;

Here, we have declared a pointer p of int type.

You can also declare pointers in these ways.

1. int \*p1;
2. int \* p2;

Let's take another example of declaring pointers.

1. int\* p1, p2;

Here, we have declared a pointer p1 and a normal variable p2.

### Assigning addresses to Pointers

Let's take an example.

1. int\* pc, c;
2. c = 5;
3. pc = &c;

Here, 5 is assigned to the c variable. And, the address of c is assigned to the pc pointer.

### Get Value of Thing Pointed by Pointers

To get the value of the thing pointed by the pointers, we use the \* operator. For example:

1. int\* pc, c;
2. c = 5;
3. pc = &c;
4. printf("%d", \*pc); // Output: 5

Here, the address of c is assigned to the pc pointer. To get the value stored in that address, we used \*pc.

**Note:**In the above example, pc is a pointer, not \*pc. You cannot and should not do something like \*pc = &c;

By the way, \* is called the dereference operator (when working with pointers). It operates on a pointer and gives the value stored in that pointer.

### Changing Value Pointed by Pointers

Let's take an example.

1. int\* pc, c;
2. c = 5;
3. pc = &c;
4. c = 1;
5. printf("%d", c); // Output: 1
6. printf("%d", \*pc); // Ouptut: 1

We have assigned the address of c to the pc pointer.

Then, we changed the value of c to 1. Since pc and the address of c is the same, \*pc gives us 1.

Let's take another example.

1. int\* pc, c;
2. c = 5;
3. pc = &c;
4. \*pc = 1;
5. printf("%d", \*pc); // Ouptut: 1
6. printf("%d", c); // Output: 1

We have assigned the address of c to the pc pointer.

Then, we changed \*pc to 1 using \*pc = 1;. Since pc and the address of c is the same, c will be equal to 1.

Let's take one more example.

1. int\* pc, c, d;
2. c = 5;
3. d = -15;
4. pc = &c; printf("%d", \*pc); // Output: 5
5. pc = &d; printf("%d", \*pc); // Ouptut: -15

Initially, the address of c is assigned to the pc pointer using pc = &c;. Since c is 5, \*pc gives us 5.

Then, the address of d is assigned to the pc pointer using pc = &d;. Since d is -15, \*pc gives us -15.

### Example: Working of Pointers

Let's take a working example.

1. #include <stdio.h>
2. int main()
3. {
4. int\* pc, c;
6. c = 22;
7. printf("Address of c: %p\n", &c);
8. printf("Value of c: %d\n\n", c); // 22
10. pc = &c;
11. printf("Address of pointer pc: %p\n", pc);
12. printf("Content of pointer pc: %d\n\n", \*pc); // 22
14. c = 11;
15. printf("Address of pointer pc: %p\n", pc);
16. printf("Content of pointer pc: %d\n\n", \*pc); // 11
18. \*pc = 2;
19. printf("Address of c: %p\n", &c);
20. printf("Value of c: %d\n\n", c); // 2
21. return 0;
22. }

**Output**

Address of c: 2686784

Value of c: 22

Address of pointer pc: 2686784

Content of pointer pc: 22

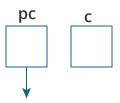
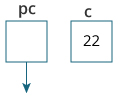
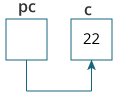
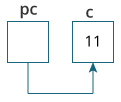
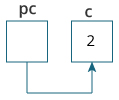
Address of pointer pc: 2686784

Content of pointer pc: 11

Address of c: 2686784

Value of c: 2

**Explanation of the program**

1. int\* pc, c;  
     
   Here, a pointer pc and a normal variable c, both of type int, is created.  
   Since pc and c are not initialized at initially, pointer pc points to either no address or a random address. And, variable c has an address but contains random garbage value.
2. c = 22;  
     
   This assigns 22 to the variable c. That is, 22 is stored in the memory location of variable c.
3. pc = &c;  
     
   This assigns the address of variable c to the pointer pc.
4. c = 11;  
     
   This assigns 11 to variable c.
5. \*pc = 2;  
     
   This change the value at the memory location pointed by the pointer pc to 2.

### Common mistakes when working with pointers

Suppose, you want pointer pc to point to the address of c. Then,

1. int c, \*pc;
2. // pc is address but c is not
3. pc = c; // Error
4. // &c is address but \*pc is not
5. \*pc = &c; // Error
6. // both &c and pc are addresses
7. pc = &c;
8. // both c and \*pc values
9. \*pc = c;

Here's an example of pointer syntax beginners often find confusing.

1. #include <stdio.h>
2. int main() {
3. int c = 5;
4. int \*p = &c;
5. printf("%d", \*p); // 5
6. return 0;
7. }

**Why didn't we get an error when using**int \*p = &c;**?**

It's because

1. int \*p = &c;

is equivalent to

1. int \*p:
2. p = &c;

In both cases, we are creating a pointer p (not \*p) and assigning &c to it.

To avoid this confusion, we can use the statement like this:

1. int\* p = &c;

# Relationship Between Arrays and Pointers

**In this tutorial, you'll learn about the relationship between arrays and pointers in C programming. You will also learn to access array elements using pointers.**

Before you learn about the relationship between arrays and pointers, be sure to check these two topics:

* [C Arrays](https://www.programiz.com/c-programming/c-arrays)
* [C Pointers](https://www.programiz.com/c-programming/c-pointers)

## Relationship Between Arrays and Pointers

An array is a block of sequential data. Let's write a program to print addresses of array elements.

1. #include <stdio.h>
2. int main()
3. {
4. int x[4];
5. int i;
6. for(i = 0; i < 4; ++i)
7. {
8. printf("&x[%d] = %p\n", i, &x[i]);
9. }
10. printf("Address of array x: %p", x);
11. return 0;
12. }

**Output**

&x[0] = 1450734448

&x[1] = 1450734452

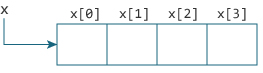
&x[2] = 1450734456

&x[3] = 1450734460

Address of array x: 1450734448

There is a difference of 4 bytes between two consecutive elements of array x. It is because the size of int is 4 bytes (on our compiler).

Notice that, the address of &x[0] and x is the same. It's because the variable name x points to the first element of the array.



From the above example, it is clear that &x[0] is equivalent to x. And, x[0] is equivalent to \*x.

Similarly,

* &x[1] is equivalent to x+1 and x[1] is equivalent to \*(x+1).
* &x[2] is equivalent to x+2 and x[2] is equivalent to \*(x+2).
* ...
* Basically, &x[i] is equivalent to x+i and x[i] is equivalent to \*(x+i).

### Example 1: Pointers and Arrays

1. #include <stdio.h>
2. int main()
3. {
4. int i, x[6], sum = 0;
5. printf("Enter 6 numbers: ");
6. for(i = 0; i < 6; ++i)
7. {
8. // Equivalent to scanf("%d", &x[i]);
9. scanf("%d", x+i);
10. // Equivalent to sum += x[i]
11. sum += \*(x+i);
12. }
13. printf("Sum = %d", sum);
14. return 0;
15. }

When you run the program, the output will be:

Enter 6 numbers: 2

3

4

4

12

4

Sum = 29

Here, we have declared an array x of 6 elements. To access elements of the array, we have used pointers.

In most contexts, array names decay to pointers. In simple words, array names are converted to pointers. That's the reason why you can use pointers to access elements of arrays. However, you should remember that **pointers and arrays are not the same**.

There are a few cases where array names don't decay to pointers. To learn more, visit: [When does array name doesn't decay into a pointer?](https://stackoverflow.com/questions/17752978/exceptions-to-array-decaying-into-a-pointer)

### Example 2: Arrays and Pointers

1. #include <stdio.h>
2. int main()
3. {
4. int x[5] = {1, 2, 3, 4, 5};
5. int\* ptr;
6. // ptr is assigned the address of the third element
7. ptr = &x[2];
8. printf("\*ptr = %d \n", \*ptr); // 3
9. printf("\*ptr+1 = %d \n", \*ptr+1); // 4
10. printf("\*ptr-1 = %d", \*ptr-1); // 2
11. return 0;
12. }

When you run the program, the output will be:

\*ptr = 3

\*ptr+1 = 4

\*ptr-1 = 2

In this example, &x[2], the address of the third element, is assigned to the ptr pointer. Hence, 3 was displayed when we printed \*ptr.

And, printing \*ptr+1 gives us the fourth element. Similarly, printing \*ptr-1 gives us the second element.

# C Call by Reference: Using pointers

**In this tutorial, you'll learn to pass addresses as arguments to the functions with the help of examples. This technique is known as call by reference.**

In C programming, it is also possible to pass addresses as arguments to functions.

To accept these addresses in the function definition, we can use pointers. It's because pointers are used to store addresses. Let's take an example:

## Example: Call by reference

1. #include <stdio.h>
2. void swap(int \*n1, int \*n2);
3. int main()
4. {
5. int num1 = 5, num2 = 10;
6. // address of num1 and num2 is passed
7. swap( &num1, &num2);
8. printf("num1 = %d\n", num1);
9. printf("num2 = %d", num2);
10. return 0;
11. }
12. void swap(int\* n1, int\* n2)
13. {
14. int temp;
15. temp = \*n1;
16. \*n1 = \*n2;
17. \*n2 = temp;
18. }

When you run the program, the output will be:

num1 = 10

num2 = 5

The address of num1 and num2 are passed to the swap() function using swap(&num1, &num2);.

Pointers n1 and n2 accept these arguments in the function definition.

1. void swap(int\* n1, int\* n2) {
2. ... ..
3. }

When \*n1 and \*n2 are changed inside the swap() function, num1 and num2 inside the main() function are also changed.

Inside the swap() function, \*n1 and \*n2 swapped. Hence, num1 and num2 are also swapped.

Notice that, swap() is not returning anything; its return type is void.

This technique is known as call by reference in C programming.

## Example 2: Passing Pointers to Functions

1. #include <stdio.h>
2. void addOne(int\* ptr) {
3. (\*ptr)++; // adding 1 to \*ptr
4. }
5. int main()
6. {
7. int\* p, i = 10;
8. p = &i;
9. addOne(p);
10. printf("%d", \*p); // 11
11. return 0;
12. }

Here, the value stored at p, \*p, is 10 initially.

We then passed the pointer p to the addOne() function. The ptr pointer gets this address in the addOne() function.

Inside the function, we increased the value stored at ptr by 1 using (\*ptr)++;. Since ptr and p pointers both have the same address, \*p inside main() is also 11.

# C File Handling

**In this tutorial, you will learn about file handling in C. You will learn to handle standard I/O in C using fprintf(), fscanf(), fread(), fwrite(), fseek() etc. with the help of examples.**

A file is a container in computer storage devices used for storing data.

## Why files are needed?

* When a program is terminated, the entire data is lost. Storing in a file will preserve your data even if the program terminates.
* If you have to enter a large number of data, it will take a lot of time to enter them all.  
  However, if you have a file containing all the data, you can easily access the contents of the file using a few commands in C.
* You can easily move your data from one computer to another without any changes.

## Types of Files

When dealing with files, there are two types of files you should know about:

1. Text files
2. Binary files

### 1. Text files

Text files are the normal **.txt** files. You can easily create text files using any simple text editors such as Notepad.

When you open those files, you'll see all the contents within the file as plain text. You can easily edit or delete the contents.

They take minimum effort to maintain, are easily readable, and provide the least security and takes bigger storage space.

### 2. Binary files

Binary files are mostly the **.bin** files in your computer.

Instead of storing data in plain text, they store it in the binary form (0's and 1's).

They can hold a higher amount of data, are not readable easily, and provides better security than text files.

## File Operations

In C, you can perform four major operations on files, either text or binary:

1. Creating a new file
2. Opening an existing file
3. Closing a file
4. Reading from and writing information to a file

## Working with files

When working with files, you need to declare a pointer of type file. This declaration is needed for communication between the file and the program.

1. FILE \*fptr;

## Opening a file - for creation and edit

Opening a file is performed using the fopen() function defined in the stdio.h header file.

The syntax for opening a file in standard I/O is:

1. ptr = fopen("fileopen","mode");

For example,

1. fopen("E:\\cprogram\\newprogram.txt","w");
2. fopen("E:\\cprogram\\oldprogram.bin","rb");

* Let's suppose the file newprogram.txt doesn't exist in the location E:\cprogram. The first function creates a new file named newprogram.txt and opens it for writing as per the mode **'w'**.  
  The writing mode allows you to create and edit (overwrite) the contents of the file.
* Now let's suppose the second binary file oldprogram.bin exists in the location E:\cprogram. The second function opens the existing file for reading in binary mode **'rb'**.  
  The reading mode only allows you to read the file, you cannot write into the file.

| Mode | Meaning of Mode | During Inexistence of file |
| --- | --- | --- |
| r | Open for reading. | If the file does not exist, fopen() returns NULL. |
| rb | Open for reading in binary mode. | If the file does not exist, fopen() returns NULL. |
| w | Open for writing. | If the file exists, its contents are overwritten. If the file does not exist, it will be created. |
| wb | Open for writing in binary mode. | If the file exists, its contents are overwritten. If the file does not exist, it will be created. |
| a | Open for append. Data is added to the end of the file. | If the file does not exist, it will be created. |
| ab | Open for append in binary mode. Data is added to the end of the file. | If the file does not exist, it will be created. |
| r+ | Open for both reading and writing. | If the file does not exist, fopen() returns NULL. |
| rb+ | Open for both reading and writing in binary mode. | If the file does not exist, fopen() returns NULL. |
| w+ | Open for both reading and writing. | If the file exists, its contents are overwritten. If the file does not exist, it will be created. |
| wb+ | Open for both reading and writing in binary mode. | If the file exists, its contents are overwritten. If the file does not exist, it will be created. |
| a+ | Open for both reading and appending. | If the file does not exist, it will be created. |
| ab+ | Open for both reading and appending in binary mode. | If the file does not exist, it will be created. |
| Opening Modes in Standard I/O | | |

## Closing a File

The file (both text and binary) should be closed after reading/writing.

Closing a file is performed using the fclose() function.

1. fclose(fptr);

Here, fptr is a file pointer associated with the file to be closed.

## Reading and writing to a text file

For reading and writing to a text file, we use the functions fprintf() and fscanf().

They are just the file versions of printf() and scanf(). The only difference is that fprint() and fscanf() expects a pointer to the structure FILE.

### Example 1: Write to a text file

1. #include <stdio.h>
2. #include <stdlib.h>
3. int main()
4. {
5. int num;
6. FILE \*fptr;
7. // use appropriate location if you are using MacOS or Linux
8. fptr = fopen("C:\\program.txt","w");
9. if(fptr == NULL)
10. {
11. printf("Error!");
12. exit(1);
13. }
14. printf("Enter num: ");
15. scanf("%d",&num);
16. fprintf(fptr,"%d",num);
17. fclose(fptr);
18. return 0;
19. }

This program takes a number from the user and stores in the file program.txt.

After you compile and run this program, you can see a text file program.txt created in C drive of your computer. When you open the file, you can see the integer you entered.

### Example 2: Read from a text file

1. #include <stdio.h>
2. #include <stdlib.h>
3. int main()
4. {
5. int num;
6. FILE \*fptr;
7. if ((fptr = fopen("C:\\program.txt","r")) == NULL){
8. printf("Error! opening file");
9. // Program exits if the file pointer returns NULL.
10. exit(1);
11. }
12. fscanf(fptr,"%d", &num);
13. printf("Value of n=%d", num);
14. fclose(fptr);
16. return 0;
17. }

This program reads the integer present in the program.txt file and prints it onto the screen.

If you successfully created the file from **Example 1**, running this program will get you the integer you entered.

Other functions like fgetchar(), fputc() etc. can be used in a similar way.

## Reading and writing to a binary file

Functions fread() and fwrite() are used for reading from and writing to a file on the disk respectively in case of binary files.

### Writing to a binary file

To write into a binary file, you need to use the fwrite() function. The functions take four arguments:

1. address of data to be written in the disk
2. size of data to be written in the disk
3. number of such type of data
4. pointer to the file where you want to write.
5. fwrite(addressData, sizeData, numbersData, pointerToFile);

### Example 3: Write to a binary file using fwrite()

1. #include <stdio.h>
2. #include <stdlib.h>
3. struct threeNum
4. {
5. int n1, n2, n3;
6. };
7. int main()
8. {
9. int n;
10. struct threeNum num;
11. FILE \*fptr;
12. if ((fptr = fopen("C:\\program.bin","wb")) == NULL){
13. printf("Error! opening file");
14. // Program exits if the file pointer returns NULL.
15. exit(1);
16. }
17. for(n = 1; n < 5; ++n)
18. {
19. num.n1 = n;
20. num.n2 = 5\*n;
21. num.n3 = 5\*n + 1;
22. fwrite(&num, sizeof(struct threeNum), 1, fptr);
23. }
24. fclose(fptr);
26. return 0;
27. }

In this program, we create a new file program.bin in the C drive.

We declare a structure threeNum with three numbers - n1, n2 and n3, and define it in the main function as num.

Now, inside the for loop, we store the value into the file using fwrite().

The first parameter takes the address of num and the second parameter takes the size of the structure threeNum.

Since we're only inserting one instance of num, the third parameter is 1. And, the last parameter \*fptr points to the file we're storing the data.

Finally, we close the file.

### Reading from a binary file

Function fread() also take 4 arguments similar to the fwrite() function as above.

1. fread(addressData, sizeData, numbersData, pointerToFile);

### Example 4: Read from a binary file using fread()

1. #include <stdio.h>
2. #include <stdlib.h>
3. struct threeNum
4. {
5. int n1, n2, n3;
6. };
7. int main()
8. {
9. int n;
10. struct threeNum num;
11. FILE \*fptr;
12. if ((fptr = fopen("C:\\program.bin","rb")) == NULL){
13. printf("Error! opening file");
14. // Program exits if the file pointer returns NULL.
15. exit(1);
16. }
17. for(n = 1; n < 5; ++n)
18. {
19. fread(&num, sizeof(struct threeNum), 1, fptr);
20. printf("n1: %d\tn2: %d\tn3: %d", num.n1, num.n2, num.n3);
21. }
22. fclose(fptr);
24. return 0;
25. }

In this program, you read the same file program.bin and loop through the records one by one.

In simple terms, you read one threeNum record of threeNum size from the file pointed by \*fptr into the structure num.

You'll get the same records you inserted in **Example 3**.

## Getting data using fseek()

If you have many records inside a file and need to access a record at a specific position, you need to loop through all the records before it to get the record.

This will waste a lot of memory and operation time. An easier way to get to the required data can be achieved using fseek().

As the name suggests, fseek() seeks the cursor to the given record in the file.

### Syntax of fseek()

1. fseek(FILE \* stream, long int offset, int whence);

The first parameter stream is the pointer to the file. The second parameter is the position of the record to be found, and the third parameter specifies the location where the offset starts.

| Whence | Meaning |
| --- | --- |
| SEEK\_SET | Starts the offset from the beginning of the file. |
| SEEK\_END | Starts the offset from the end of the file. |
| SEEK\_CUR | Starts the offset from the current location of the cursor in the file. |
| Different whence in fseek() | |

### Example 5: fseek()

1. #include <stdio.h>
2. #include <stdlib.h>
3. struct threeNum
4. {
5. int n1, n2, n3;
6. };
7. int main()
8. {
9. int n;
10. struct threeNum num;
11. FILE \*fptr;
12. if ((fptr = fopen("C:\\program.bin","rb")) == NULL){
13. printf("Error! opening file");
14. // Program exits if the file pointer returns NULL.
15. exit(1);
16. }
18. // Moves the cursor to the end of the file
19. fseek(fptr, -sizeof(struct threeNum), SEEK\_END);
20. for(n = 1; n < 5; ++n)
21. {
22. fread(&num, sizeof(struct threeNum), 1, fptr);
23. printf("n1: %d\tn2: %d\tn3: %d\n", num.n1, num.n2, num.n3);
24. fseek(fptr, -2\*sizeof(struct threeNum), SEEK\_CUR);
25. }
26. fclose(fptr);
28. return 0;
29. }

This program will start reading the records from the file program.bin in the reverse order (last to first) and prints it.

# C Files Examples

**In this article, you'll find a list of examples to handle file input/output operations in C programming.**

To understand all programs on this page, you should have the knowledge of the following topics.

* [C Arrays](https://www.programiz.com/c-programming/c-arrays)
* [C Pointers](https://www.programiz.com/c-programming/c-pointers)
* [Array and Pointer Relation](https://www.programiz.com/c-programming/c-pointers-arrays)
* [File I/O](https://www.programiz.com/c-programming/c-file-input-output)

## C File Examples

**1. C program to read name and marks of n number of students and store them in a file.**

1. #include <stdio.h>
2. int main()
3. {
4. char name[50];
5. int marks, i, num;
6. printf("Enter number of students: ");
7. scanf("%d", &num);
8. FILE \*fptr;
9. fptr = (fopen("C:\\student.txt", "w"));
10. if(fptr == NULL)
11. {
12. printf("Error!");
13. exit(1);
14. }
15. for(i = 0; i < num; ++i)
16. {
17. printf("For student%d\nEnter name: ", i+1);
18. scanf("%s", name);
19. printf("Enter marks: ");
20. scanf("%d", &marks);
21. fprintf(fptr,"\nName: %s \nMarks=%d \n", name, marks);
22. }
23. fclose(fptr);
24. return 0;
25. }

**2. C program to read name and marks of n number of students from and store them in a file. If the file previously exits, add the information to the file.**

1. #include <stdio.h>
2. int main()
3. {
4. char name[50];
5. int marks, i, num;
6. printf("Enter number of students: ");
7. scanf("%d", &num);
8. FILE \*fptr;
9. fptr = (fopen("C:\\student.txt", "a"));
10. if(fptr == NULL)
11. {
12. printf("Error!");
13. exit(1);
14. }
15. for(i = 0; i < num; ++i)
16. {
17. printf("For student%d\nEnter name: ", i+1);
18. scanf("%s", name);
19. printf("Enter marks: ");
20. scanf("%d", &marks);
21. fprintf(fptr,"\nName: %s \nMarks=%d \n", name, marks);
22. }
23. fclose(fptr);
24. return 0;
25. }

**3. C program to write all the members of an array of structures to a file using fwrite(). Read the array from the file and display on the screen.**

1. #include <stdio.h>
2. struct student
3. {
4. char name[50];
5. int height;
6. };
7. int main(){
8. struct student stud1[5], stud2[5];
9. FILE \*fptr;
10. int i;
11. fptr = fopen("file.txt","wb");
12. for(i = 0; i < 5; ++i)
13. {
14. fflush(stdin);
15. printf("Enter name: ");
16. gets(stud1[i].name);
17. printf("Enter height: ");
18. scanf("%d", &stud1[i].height);
19. }
20. fwrite(stud1, sizeof(stud1), 1, fptr);
21. fclose(fptr);
22. fptr = fopen("file.txt", "rb");
23. fread(stud2, sizeof(stud2), 1, fptr);
24. for(i = 0; i < 5; ++i)
25. {
26. printf("Name: %s\nHeight: %d", stud2[i].name, stud2[i].height);
27. }