C

Programming

Table of Content

C

C is a powerful mid to low-level compiled programming language used in operating systems, as the base for higher level languages like C++ and Python, and in high-performance applications. It excels in speed and performance giving the programmer great control over the system.

The C language was created by Dennis Ritchie in Bell Labs in the 1970s. It was designed to be a system implementation language for the nascent Unix operating system.
C is strongly associated with UNIX, as it was developed to write the UNIX operating system.

Why Learn C?

- It is one of the most popular programming language in the world
- If you know C, you will have no problem to learn other popular programming languages such as Java, Python, C++, C#, etc, as the syntax is similar
- C is very fast, compared to other programming languages, like Java and Python
- C is very versatile; it can be used in both applications and technologies

---

Difference between C and C++

- C++ was developed as an extension of C, and both languages have almost the same syntax
- The main difference between C and C++ is that C++ support classes and objects, while C does not

C Keywords and Identifiers

In this tutorial, you will learn about keywords; reserved words in C programming that are part of the syntax. Also, you will learn about identifiers and how to name them.

Character set

A character set is a set of alphabets, letters and some special characters that are valid in C language.

Alphabets

```
Uppercase: A B C ................................... X Y Z
Lowercase: a b c ...................................... x y z
```

C accepts both lowercase and uppercase alphabets as variables and functions.

Digits

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

Special Characters

,	<	>	.	_
(	)	;	$	:

%	[	]	#	?
'	&	{	}	"
^	!	*	/	|
-	\	~	+	

White space Characters

Blank space, newline, horizontal tab, carriage return and form feed.

C Keywords

Keywords are predefined, reserved words used in programming that have special meanings to the compiler. Keywords are part of the syntax and they cannot be used as an identifier. For example:

```
int money;
```

Here, `int` is a keyword that indicates `money` is a variable of type `int` (integer).

As C is a case sensitive language, all keywords must be written in lowercase. Here is a list of all keywords allowed in ANSI C.

auto	double	int	struct

break	else	long	switch
case	enum	register	typedef
char	extern	return	union
continue	for	signed	void
do	if	static	while
default	goto	sizeof	volatile
const	float	short	unsigned

All these keywords, their syntax, and application will be discussed in their respective topics.

C Identifiers

Identifier refers to name given to entities such as variables, functions, structures etc.

Identifiers must be unique. They are created to give a unique name to an entity to identify it during the execution of the program. For example:

```
int money;
double accountBalance;
```

Here, `money` and `accountBalance` are identifiers.

Also remember, identifier names must be different from keywords. You cannot use `int` as an identifier because `int` is a keyword.

Rules for naming identifiers

1. A valid identifier can have letters (both uppercase and lowercase letters), digits and underscores.
2. The first letter of an identifier should be either a letter or an underscore.
3. You cannot use keywords like `int`, `while` etc. as identifiers.
4. There is no rule on how long an identifier can be. However, you may run into problems in some compilers if the identifier is longer than 31 characters.

You can choose any name as an identifier if you follow the above rule, however, give meaningful names to identifiers that make sense.

C Variables, Constants and Literals

Variables

In programming, a variable is a container (storage area) to hold data.

To indicate the storage area, each variable should be given a unique name (identifier). Variable names are just the symbolic representation of a memory location. For example:

```
int playerScore = 95;
```

Here, `playerScore` is a variable of `int` type. Here, the variable is assigned an integer value `95`.

The value of a variable can be changed, hence the name variable.

```
char ch = 'a';
// some code
ch = 'l';
```

Rules for naming a variable

1. A variable name can only have letters (both uppercase and lowercase letters), digits and underscore.
2. The first letter of a variable should be either a letter or an underscore.
3. There is no rule on how long a variable name (identifier) can be. However, you may run into problems in some compilers if the variable name is longer than 31 characters.

Note: You should always try to give meaningful names to variables. For example: `firstName` is a better variable name than `fn`.

C is a strongly typed language. This means that the variable type cannot be changed once it is declared. For example:

```
int number = 5;        // integer variable
number = 5.5;          // error
double number;         // error
```

Here, the type of `number` variable is `int`. You cannot assign a floating-point (decimal) value `5.5` to this variable. Also, you cannot redefine the data type of the variable to `double`. By the way, to store the decimal values in C, you need to declare its type to either `double` or `float`.

Literals

Literals are data used for representing fixed values. They can be used directly in the code. For example: `1`, `2.5`, `'c'` etc.

Here, `1`, `2.5` and `'c'` are literals. Why? You cannot assign different values to these terms.

---

1. Integers

An integer is a numeric literal(associated with numbers) without any fractional or exponential part. There are three types of integer literals in C programming:

- decimal (base 10)

- octal (base 8)
- hexadecimal (base 16)

For example:

```
Decimal: 0, -9, 22 etc
Octal: 021, 077, 033 etc
Hexadecimal: 0x7f, 0x2a, 0x521 etc
```

In C programming, octal starts with a `0`, and hexadecimal starts with a `0x`.

---

2. Floating-point Literals

A floating-point literal is a numeric literal that has either a fractional form or an exponent form. For example:

```
-2.0
0.0000234
-0.22E-5
```

> Note: `E-5 = 10-5`

---

3. Characters

A character literal is created by enclosing a single character inside single quotation marks. For example: `'a'`, `'m'`, `'F'`, `'2'`, `'}'` etc.

---

4. Escape Sequences

Sometimes, it is necessary to use characters that cannot be typed or has special meaning in C programming. For example: newline(enter), tab, question mark etc.

In order to use these characters, escape sequences are used.

Escape Sequences	Character
`\b`	Backspace
`\f`	Form feed
`\n`	Newline
`\r`	Return
`\t`	Horizontal tab
`\v`	Vertical tab
`\\`	Backslash
`\'`	Single quotation mark

`\"`	Double quotation mark
`\?`	Question mark
`\0`	Null character

For example: `\n` is used for a newline. The backslash `\` causes escape from the normal way the characters are handled by the compiler.

5. String Literals

A string literal is a sequence of characters enclosed in double-quote marks. For example:

```
"good"                    //string constant
""                       //null string constant
"      "                 //string constant of six white space
"x"                      //string constant having a single character.
"Earth is round\n"           //prints string with a newline
```

Constants

If you want to define a variable whose value cannot be changed, you can use the `const` keyword. This will create a constant. For example,

```
const double PI = 3.14;
```

Notice, we have added keyword `const`.

Here, `PI` is a symbolic constant; its value cannot be changed.

```
const double PI = 3.14;
PI = 2.9; //Error
```

C Data Types

In C programming, data types are declarations for variables. This determines the type and size of data associated with variables. For example,

```
int myVar;
```

Here, `myVar` is a variable of `int` (integer) type. The size of `int` is 4 bytes.

Basic types

Here's a table containing commonly used types in C programming for quick access.

Type	Size (bytes)	Format Specifier
`int`	at least 2, usually 4	`%d`, `%i`
`char`	1	`%c`
`float`	4	`%f`
`double`	8	`%lf`

`short int`	2 usually	`%hd`
`unsigned int`	at least 2, usually 4	`%u`
`long int`	at least 4, usually 8	`%ld`, `%li`
`long long int`	at least 8	`%lld`, `%lli`
`unsigned long int`	at least 4	`%lu`
`unsigned long long int`	at least 8	`%llu`
`signed char`	1	`%c`
`unsigned char`	1	`%c`
`long double`	at least 10, usually 12 or 16	`%Lf`

int

Integers are whole numbers that can have both zero, positive and negative values but no decimal values. For example, `0`, `-5`, `10`

We can use `int` for declaring an integer variable.

```
int id;
```

Here, `id` is a variable of type integer.

You can declare multiple variables at once in C programming. For example,

```
int id, age;
```

The size of `int` is usually 4 bytes (32 bits). And, it can take `232` distinct states from `-2147483648` to `2147483647`.

---

float and double

`float` and `double` are used to hold real numbers.

```
float salary;
double price;
```

In C, floating-point numbers can also be represented in exponential. For example,

```
float normalizationFactor = 22.442e2;
```

What's the difference between `float` and `double`?

The size of `float` (single precision float data type) is 4 bytes. And the size of `double` (double precision float data type) is 8 bytes.

---

char

Keyword `char` is used for declaring character type variables. For example,

```
char test = 'h';
```

The size of the character variable is 1 byte.

void

`void` is an incomplete type. It means "nothing" or "no type". You can think of void as absent.

For example, if a function is not returning anything, its return type should be `void`.

Note that, you cannot create variables of `void` type.

short and long

If you need to use a large number, you can use a type specifier `long`. Here's how:

```
long a;
long long b;
long double c;
```

Here variables `a` and `b` can store integer values. And, `c` can store a floating-point number.

If you are sure, only a small integer (`[-32,767, +32,767]` range) will be used, you can use `short`.

```
short d;
```

You can always check the size of a variable using the `sizeof()` operator.

```c
#include <stdio.h>
int main() {
  short a;
  long b;
  long long c;
  long double d;

  printf("size of short = %d bytes\n", sizeof(a));
  printf("size of long = %d bytes\n", sizeof(b));
  printf("size of long long = %d bytes\n", sizeof(c));
  printf("size of long double= %d bytes\n", sizeof(d));
  return 0;
}
```

signed and unsigned

In C, `signed` and `unsigned` are type modifiers. You can alter the data storage of a data type by using them:

- `signed` - allows for storage of both positive and negative numbers
- `unsigned` - allows for storage of only positive numbers

For example,

```c
// valid codes
unsigned int x = 35;
int y = -35;  // signed int
int z = 36;  // signed int

// invalid code: unsigned int cannot hold negative integers
unsigned int num = -35;
```

Here, the variables `x` and `num` can hold only zero and positive values because we have used the `unsigned` modifier.

Considering the size of `int` is 4 bytes, variable `y` can hold values from -2^{31} to $2^{31}-1$, whereas variable `x` can hold values from `0` to $2^{32}-1$.

C Input Output (I/O)

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

```
#include <stdio.h>
int main()
{
    // Displays the string inside quotations
    printf("C Programming");
    return 0;
}
```

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 the `#include <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

```c
#include <stdio.h>
int main()
{
    int testInteger = 5;
    printf("Number = %d", testInteger);
    return 0;
}
```

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

```c
#include <stdio.h>
int main()
{
    float number1 = 13.5;
    double number2 = 12.4;

    printf("number1 = %f\n", number1);
    printf("number2 = %lf", number2);
    return 0;
}
```

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

```c
#include <stdio.h>
int main()
{
    char chr = 'a';
    printf("character = %c", chr);
    return 0;
}
```

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

```c
#include <stdio.h>
int main()
{
    int testInteger;
    printf("Enter an integer: ");
    scanf("%d", &testInteger);
    printf("Number = %d",testInteger);
    return 0;
}
```

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

```c
#include <stdio.h>
int main()
{
    float num1;
    double num2;

    printf("Enter a number: ");
    scanf("%f", &num1);
    printf("Enter another number: ");
    scanf("%lf", &num2);

    printf("num1 = %f\n", num1);
    printf("num2 = %lf", num2);

    return 0;
}
```

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

```
#include <stdio.h>
int main()
{
    char chr;
    printf("Enter a character: ");
    scanf("%c",&chr);
    printf("You entered %c.", chr);
    return 0;
}
```

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

```
#include <stdio.h>
int main()
{
    char chr;
    printf("Enter a character: ");
    scanf("%c", &chr);

    // When %c is used, a character is displayed
    printf("You entered %c.\n",chr);

    // When %d is used, ASCII value is displayed
    printf("ASCII value is %d.", chr);
    return 0;
}
```

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.

```
#include <stdio.h>
int main()
{
    int a;
    float b;

    printf("Enter integer and then a float: ");

    // Taking multiple inputs
    scanf("%d%f", &a, &b);

    printf("You entered %d and %f", a, b);
    return 0;
}
```

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 Comments

In programming, comments are hints that a programmer can add to make their code easier to read and understand. For example,

```
#include <stdio.h>

int main() {

  // print Hello World to the screen
  printf("Hello World");
  return 0;
}
```

Output

```
Hello World
```

Here, `// print Hello World to the screen` is a comment in C programming. Comments are completely ignored by C compilers.

Types of Comments

There are two ways to add comments in C:

1. `//` - Single Line Comment
2. `/*...*/` - Multi-line Comment

---

1. Single-line Comments in C

In C, a single line comment starts with `//`. It starts and ends in the same line.

For example,

```
#include <stdio.h>

int main() {

  // create integer variable
  int age = 25;

  // print the age variable
  printf("Age: %d", age);

  return 0;
}
```

Output

```
Age: 25
```

In the above example, `// create integer variable` and `// print the age variable` are two single line comments.

We can also use the single line comment along with the code. For example,

```
int age = 25;   // create integer variable
```

Here, code before `//` are executed and code after `//` are ignored by the compiler.

---

2. Multi-line Comments in C

In C programming, there is another type of comment that allows us to comment on multiple lines at once, they are multi-line comments.

To write multi-line comments, we use the `/*....*/` symbol. For example,

```
/* This program takes age input from the user
It stores it in the age variable
And, print the value using printf() */

#include <stdio.h>

int main() {

  int age;

  printf("Enter the age: ");
  scanf("%d", &age);

  printf("Age = %d", age);

  return 0;
}
```

Output

```
Enter the age: 24
Age = 24
```

In this type of comment, the C compiler ignores everything from `/*` to `*/`.

Note: Remember the keyboard shortcut to use comments:

- Single Line comment: ctrl + / (windows) and cmd + / (mac)
- Multi line comment: ctrl + shift + / (windows) and cmd + shift + / (mac)

Use of Comments in C

1. Make Code Easier to Understand

If we write comments on our code, it will be easier to understand the code in the future. Otherwise you will end up spending a lot of time looking at our own code and trying to understand it.

Comments are even more important if you are working in a group. It makes it easier for other developers to understand and use your code.

2. Using Comments for debugging

While debugging there might be situations where we don't want some part of the code. For example,

In the program below, suppose we don't need data related to height. So, instead of removing the code related to height, we can simply convert them into comments.

```
// Program to take age and height input

#include <stdio.h>

int main() {

  int age;
  // double height;

  printf("Enter the age: ");
  scanf("%d", &age);

  // printf("Enter the height: ");
  // scanf("%lf", &height);

  printf("Age = %d", age);
  // printf("\nHeight = %.1lf", height);

  return 0;
}
```

Now later on, if we need height again, all you need to do is remove the forward slashes. And, they will now become statements not comments.

Note: Comments are not and should not be used as a substitute to explain poorly written code. Always try to write clean, understandable code, and then use comments as an addition.

In most cases, always use comments to explain 'why' rather than 'how' and you are good to go.

C Programming Operators

An operator is a symbol that operates on a value or a variable. For example: + is an operator to perform addition.

C has a wide range of operators to perform various operations.

C Arithmetic Operators

An arithmetic operator performs mathematical operations such as addition, subtraction, multiplication, division etc on numerical values (constants and variables).

Operator	Meaning of Operator
+	addition or unary plus
-	subtraction or unary minus
*	multiplication
/	division

%	remainder after division (modulo division)

Example 1: Arithmetic Operators

```
// Working of arithmetic operators
#include <stdio.h>
int main()
{
    int a = 9,b = 4, c;

    c = a+b;
    printf("a+b = %d \n",c);
    c = a-b;
    printf("a-b = %d \n",c);
    c = a*b;
    printf("a*b = %d \n",c);
    c = a/b;
    printf("a/b = %d \n",c);
    c = a%b;
    printf("Remainder when a divided by b = %d \n",c);

    return 0;
}
```

Output

```
a+b = 13
a-b = 5
a*b = 36
a/b = 2
Remainder when a divided by b=1
```

The operators `+`, `-` and `*` computes addition, subtraction, and multiplication respectively as you might have expected.

In normal calculation, `9/4 = 2.25`. However, the output is `2` in the program.

It is because both the variables `a` and b are integers. Hence, the output is also an integer. The compiler neglects the term after the decimal point and shows answer `2` instead of `2.25`.

The modulo operator `%` computes the remainder. When `a=9` is divided by `b=4`, the remainder is `1`. The `%` operator can only be used with integers.

Suppose `a = 5.0`, `b = 2.0`, `c = 5` and `d = 2`. Then in C programming,

```
// Either one of the operands is a floating-point number
a/b = 2.5
a/d = 2.5
c/b = 2.5

// Both operands are integers
c/d = 2
```

C Increment and Decrement Operators

C programming has two operators increment `++` and decrement `--` to change the value of an operand (constant or variable) by 1.

Increment `++` increases the value by 1 whereas decrement `--` decreases the value by 1. These two operators are unary operators, meaning they only operate on a single operand.

Example 2: Increment and Decrement Operators

```
// Working of increment and decrement operators
#include <stdio.h>
int main()
{
    int a = 10, b = 100;
    float c = 10.5, d = 100.5;
    printf("++a = %d \n", ++a);
    printf("--b = %d \n", --b);
    printf("++c = %f \n", ++c);
    printf("--d = %f \n", --d);

    return 0;
}
```

Output

```
++a = 11
--b = 99
++c = 11.500000
--d = 99.500000
```

Here, the operators `++` and `--` are used as prefixes. These two operators can also be used as postfixes like `a++` and `a--`. Visit this page to learn more about how increment and decrement operators work when used as postfix.

C Assignment Operators

An assignment operator is used for assigning a value to a variable. The most common assignment operator is `=`

Operator	Example	Same as
=	a = b	a = b
+=	a += b	a = a+b

-=	a -= b	a = a-b
*=	a *= b	a = a*b
/=	a /= b	a = a/b
%=	a %= b	a = a%b

Example 3: Assignment Operators

```
// Working of assignment operators
#include <stdio.h>
int main()
{
    int a = 5, c;

    c = a;      // c is 5
    printf("c = %d\n", c);
    c += a;     // c is 10
    printf("c = %d\n", c);
    c -= a;     // c is 5
    printf("c = %d\n", c);
    c *= a;     // c is 25
    printf("c = %d\n", c);
    c /= a;     // c is 5
    printf("c = %d\n", c);
    c %= a;     // c = 0
    printf("c = %d\n", c);

    return 0;
}
```

Output

```
c = 5
c = 10
c = 5
```

```
c = 25
c = 5
c = 0
```

C Relational Operators

A relational operator checks the relationship between two operands. If the relation is true, it returns 1; if the relation is false, it returns value 0.

Relational operators are used in decision making and loops.

Operator	Meaning of Operator	Example
==	Equal to	`5 == 3` is evaluated to 0
>	Greater than	`5 > 3` is evaluated to 1
<	Less than	`5 < 3` is evaluated to 0
!=	Not equal to	`5 != 3` is evaluated to 1
>=	Greater than or equal to	`5 >= 3` is evaluated to 1
<=	Less than or equal to	`5 <= 3` is evaluated to 0

Example 4: Relational Operators

```
// Working of relational operators
```

```
#include <stdio.h>
int main()
{
    int a = 5, b = 5, c = 10;

    printf("%d == %d is %d \n", a, b, a == b);
    printf("%d == %d is %d \n", a, c, a == c);
    printf("%d > %d is %d \n", a, b, a > b);
    printf("%d > %d is %d \n", a, c, a > c);
    printf("%d < %d is %d \n", a, b, a < b);
    printf("%d < %d is %d \n", a, c, a < c);
    printf("%d != %d is %d \n", a, b, a != b);
    printf("%d != %d is %d \n", a, c, a != c);
    printf("%d >= %d is %d \n", a, b, a >= b);
    printf("%d >= %d is %d \n", a, c, a >= c);
    printf("%d <= %d is %d \n", a, b, a <= b);
    printf("%d <= %d is %d \n", a, c, a <= c);

    return 0;
}
```

Output

```
5 == 5 is 1
5 == 10 is 0
5 > 5 is 0
5 > 10 is 0
5 < 5 is 0
5 < 10 is 1
5 != 5 is 0
5 != 10 is 1
5 >= 5 is 1
5 >= 10 is 0
5 <= 5 is 1
5 <= 10 is 1
```

C Logical Operators

An expression containing logical operator returns either 0 or 1 depending upon whether expression results true or false. Logical operators are commonly used in decision making in C programming.

Operator	Meaning	Example
&&	Logical AND. True only if all operands are true	If c = 5 and d = 2 then, expression `((c==5) && (d>5))` equals to 0.
||	Logical OR. True only if either one operand is true	If c = 5 and d = 2 then, expression `((c==5) || (d>5))` equals to 1.
!	Logical NOT. True only if the operand is 0	If c = 5 then, expression `!(c==5)` equals to 0.

Example 5: Logical Operators

```
// Working of logical operators

#include <stdio.h>
int main()
{
    int a = 5, b = 5, c = 10, result;

    result = (a == b) && (c > b);
    printf("(a == b) && (c > b) is %d \n", result);

    result = (a == b) && (c < b);
    printf("(a == b) && (c < b) is %d \n", result);

    result = (a == b) || (c < b);
    printf("(a == b) || (c < b) is %d \n", result);

    result = (a != b) || (c < b);
    printf("(a != b) || (c < b) is %d \n", result);

    result = !(a != b);
    printf("!(a != b) is %d \n", result);

    result = !(a == b);
    printf("!(a == b) is %d \n", result);

    return 0;
}
```

Output

```
(a == b) && (c > b) is 1
(a == b) && (c < b) is 0
(a == b) || (c < b) is 1
(a != b) || (c < b) is 0
!(a != b) is 1
!(a == b) is 0
```

Explanation of logical operator program

- `(a == b) && (c > 5)` evaluates to 1 because both operands `(a == b)` and `(c > b)` is 1 (true).
- `(a == b) && (c < b)` evaluates to 0 because operand `(c < b)` is 0 (false).
- `(a == b) || (c < b)` evaluates to 1 because `(a = b)` is 1 (true).
- `(a != b) || (c < b)` evaluates to 0 because both operand `(a != b)` and `(c < b)` are 0 (false).
- `!(a != b)` evaluates to 1 because operand `(a != b)` is 0 (false). Hence, !(a != b) is 1 (true).
- `!(a == b)` evaluates to 0 because `(a == b)` is 1 (true). Hence, `!(a == b)` is 0 (false).

C Bitwise Operators

During computation, mathematical operations like: addition, subtraction, multiplication, division, etc are converted to bit-level which makes processing faster and saves power.

Bitwise operators are used in C programming to perform bit-level operations.

Operators	Meaning of operators
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
~	Bitwise complement
<<	Shift left
>>	Shift right

Visit bitwise operator in C to learn more.

Other Operators

Comma Operator

Comma operators are used to link related expressions together. For example:

```
int a, c = 5, d;
```

The sizeof operator

The `sizeof` is a unary operator that returns the size of data (constants, variables, array, structure, etc).

Example 6: sizeof Operator

```
#include <stdio.h>
int main()
{
    int a;
    float b;
    double c;
    char d;
    printf("Size of int=%lu bytes\n",sizeof(a));
    printf("Size of float=%lu bytes\n",sizeof(b));
    printf("Size of double=%lu bytes\n",sizeof(c));
    printf("Size of char=%lu byte\n",sizeof(d));

    return 0;
}
```

Output

```
Size of int = 4 bytes
Size of float = 4 bytes
Size of double = 8 bytes
Size of char = 1 byte
```

Examples

C program to print a sentence

C program to print an integer entered by the user

C program to add two integers entered by the User

C program to multiply two floating-point numbers

C program to find ASCII value of a character entered by the user

---

C program to find quotient and remainder of Two Integers

C program to find the size of int, float, double and char

C program to demonstrate the working of keyword long

C program to swap two numbers

C if...else Statement

C if Statement

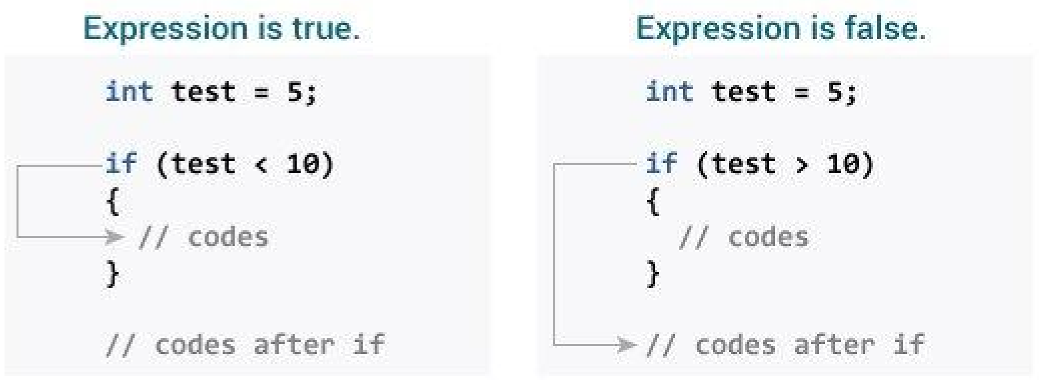
The syntax of the `if` statement in C programming is:

```
if (test expression)
{
   // code
}
```

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.



```
int test = 5;

if (test < 10)
{
   // codes
}

// codes after if
```

```
int test = 5;

if (test > 10)
{
   // codes
}

// codes after if
```

Working of if Statement

To learn more about when test expression is evaluated to true (non-zero value) and false (0), check relational and logical operators.

Example 1: if statement

```c
// Program to display a number if it is negative

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

    printf("Enter an integer: ");
    scanf("%d", &number);

    // true if number is less than 0
    if (number < 0) {
        printf("You entered %d.\n", number);
    }

    printf("The if statement is easy.");

    return 0;
}
```

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:

```
if (test expression) {
    // run code if test expression is true
}
else {
    // run code if test expression is false
}
```

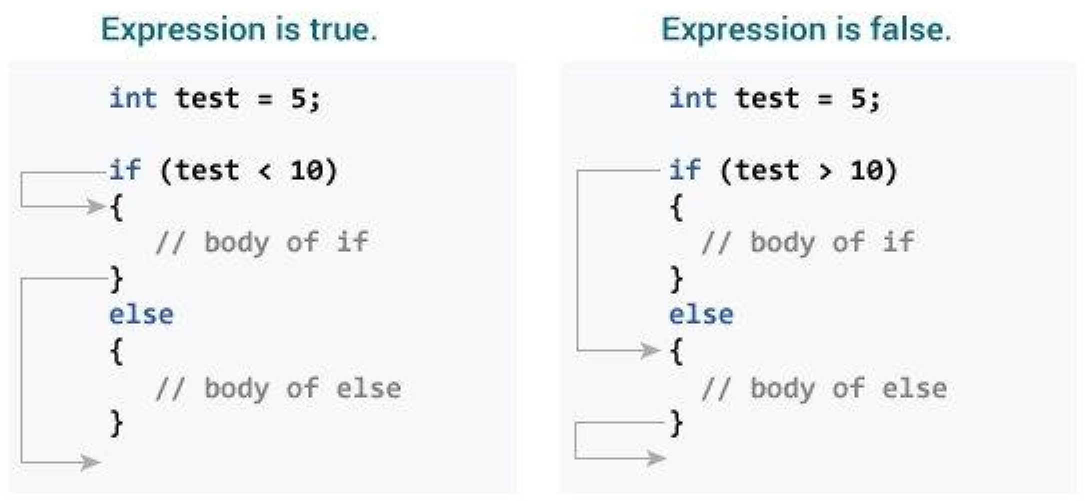
How if...else statement works?

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.



```
int test = 5;

if (test < 10)
{
   // body of if
}
else
{
   // body of else
}
```

```
int test = 5;

if (test > 10)
{
   // body of if
}
else
{
   // body of else
}
```

Working of if...else Statement

Example 2: if...else statement

```
// Check whether an integer is odd or even

#include <stdio.h>
int main() {
    int number;
    printf("Enter an integer: ");
    scanf("%d", &number);

    // True if the remainder is 0
    if  (number%2 == 0) {
        printf("%d is an even integer.",number);
    }
    else {
        printf("%d is an odd integer.",number);
    }

    return 0;
}
```

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 if...else Ladder

```
if (test expression1) {
   // statement(s)
}
else if(test expression2) {
   // statement(s)
}
else if (test expression3) {
   // statement(s)
}
.
.
else {
   // statement(s)
}
```

Example 3: C if...else Ladder

```
// Program to relate two integers using =, > or < symbol

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

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

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

    //checks if both test expressions are false
    else {
        printf("Result: %d < %d",number1, number2);
    }

    return 0;
}
```

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.

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

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

    return 0;
}
```

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

For example, this code

```c
if (a > b) {
    printf("Hello");
}
printf("Hi");
```

is equivalent to

```c
if (a > b)
    printf("Hello");
printf("Hi");
```

C for Loop

In programming, a loop is used to repeat a block of code until the 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:

```
for (initializationStatement; testExpression; updateStatement)
{
    // statements inside the body of loop
}
```

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.

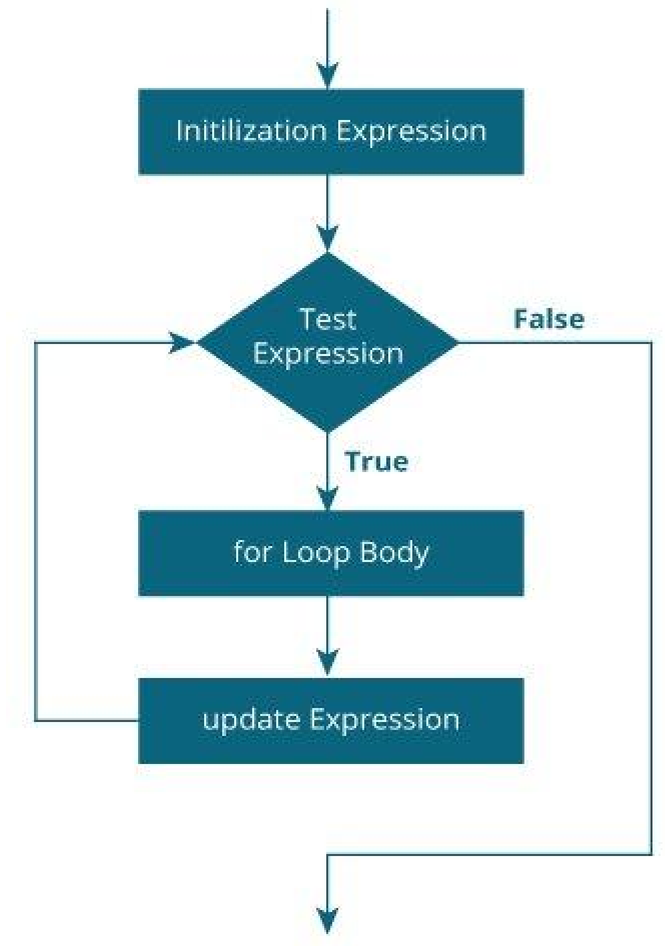
- However, if the test expression is evaluated to true, statements inside the body of the `for` loop are executed, and the update expression is updated.
- Again the test expression is evaluated.

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 and logical operators.

---

for loop Flowchart



Working of for loop

Example 1: for loop

```
// Print numbers from 1 to 10
#include <stdio.h>

int main() {
  int i;

  for (i = 1; i < 11; ++i)
  {
    printf("%d ", i);
  }
  return 0;
}
```

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

```c
// Program to calculate the sum of first n natural numbers
// Positive integers 1,2,3...n are known as natural numbers

#include <stdio.h>
int main()
{
    int num, count, sum = 0;

    printf("Enter a positive integer: ");
    scanf("%d", &num);

    // for loop terminates when num is less than count
    for(count = 1; count <= num; ++count)
    {
        sum += count;
    }

    printf("Sum = %d", sum);
    return 0;
}
```

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 `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 the `for` loop is executed. Now, `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 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:

```
while (testExpression) {
  // the body of the loop
}
```

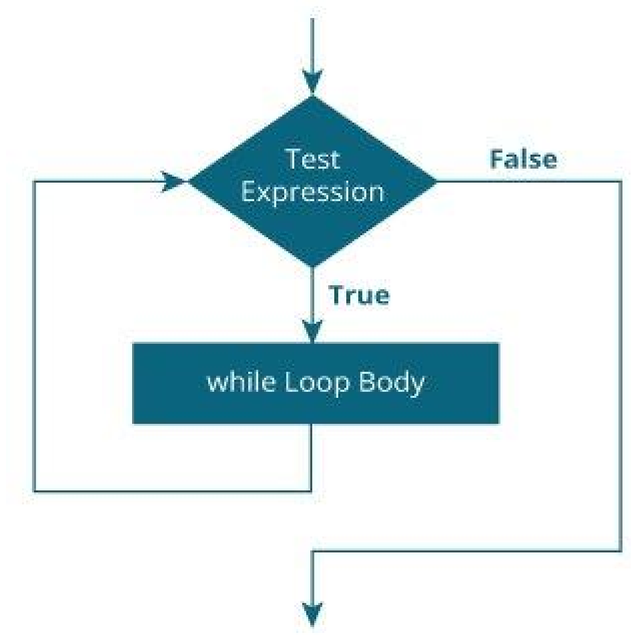
How while loop works?

- The `while` loop evaluates the `testExpression` inside the parentheses `()`.
- If `testExpression` is true, statements inside the body of `while` loop are executed. Then, `testExpression` is evaluated again.

- The process goes on until `testExpression` is evaluated to false.
- If `testExpression` is false, the loop terminates (ends).

To learn more about test expressions (when `testExpression` is evaluated to true and false), check out relational and logical operators.

Flowchart of while loop



Working of while loop

Example 1: while loop

```
// Print numbers from 1 to 5

#include <stdio.h>
int main() {
  int i = 1;

  while (i <= 5) {
    printf("%d\n", i);
    ++i;
  }

  return 0;
}
```

Output

```
1
2
3
4
5
```

Here, we have initialized `i` to 1.

1. When `i = 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 = 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. Then, 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.

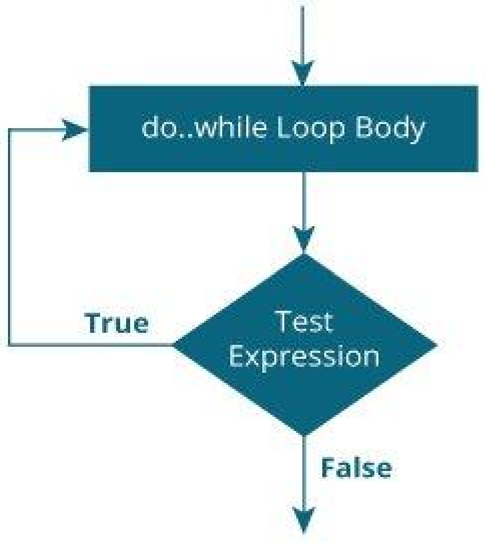
The syntax of the `do...while` loop is:

```
do {
  // the body of the loop
}
while (testExpression);
```

How do...while loop works?

- The body of `do...while` loop is executed once. Only then, the `testExpression` is evaluated.
- If `testExpression` is true, the body of the loop is executed again and `testExpression` is evaluated once more.
- This process goes on until `testExpression` becomes false.
- If `testExpression` is false, the loop ends.

Flowchart of do...while Loop



Working of do...while loop

Example 2: do...while loop

```
// Program to add numbers until the user enters zero

#include <stdio.h>
int main() {
  double number, sum = 0;

  // the body of the loop is executed at least once
  do {
    printf("Enter a number: ");
    scanf("%lf", &number);
    sum += number;
  }
  while(number != 0.0);

  printf("Sum = %.2lf",sum);

  return 0;
}
```

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
```

Here, we have used a `do...while` loop to prompt the user to enter a number. The loop works as long as the input number is not `0`.

The `do...while` loop executes at least once i.e. the first iteration runs without checking the condition. The condition is checked only after the first iteration has been executed.

```
do {
  printf("Enter a number: ");
  scanf("%lf", &number);
```

```
   sum += number;
}
while(number != 0.0);
```

So, if the first input is a non-zero number, that number is added to the `sum` variable and the loop continues to the next iteration. This process is repeated until the user enters `0`.

But if the first input is 0, there will be no second iteration of the loop and `sum` becomes `0.0`.

Outside the loop, we print the value of `sum`.

C break and continue

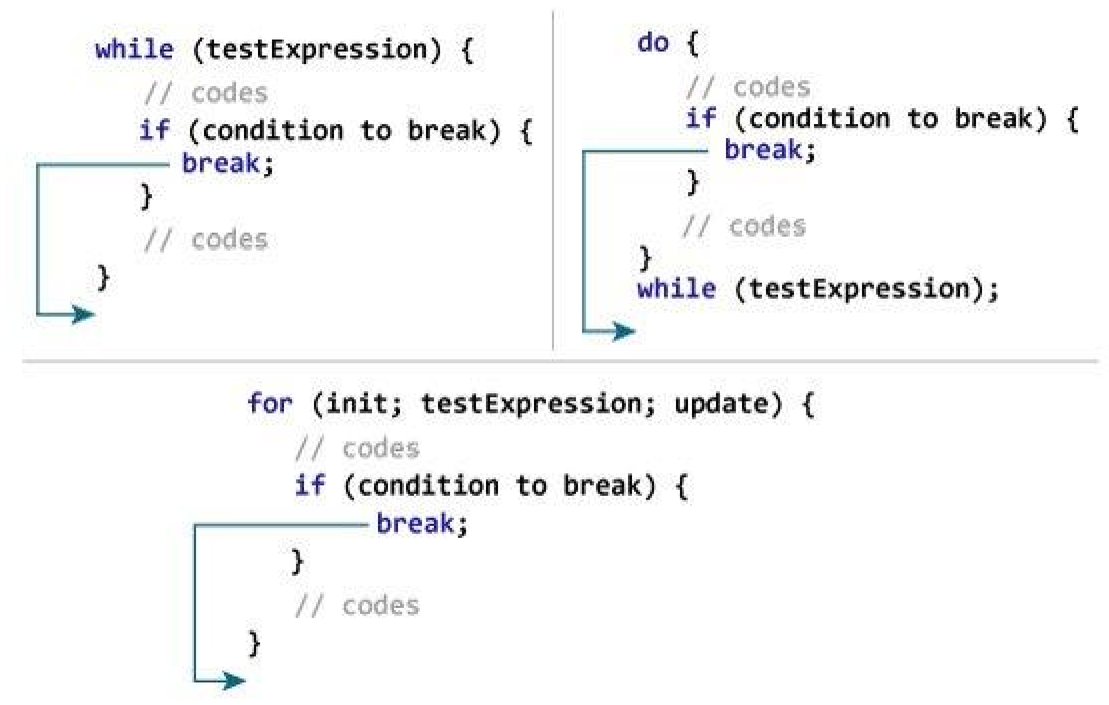
C break

The break statement ends the loop immediately when it is encountered. Its syntax is:

```
break;
```

The break statement is almost always used with `if...else` statement inside the loop.

How break statement works?



```
while (testExpression) {
    // codes
    if (condition to break) {
        break;
    }
    // codes
}
```

```
do {
    // codes
    if (condition to break) {
        break;
    }
    // codes
}
while (testExpression);
```

```
for (init; testExpression; update) {
    // codes
    if (condition to break) {
        break;
    }
    // codes
}
```

Working of break in C

Example 1: break statement

```
// Program to calculate the sum of numbers (10 numbers max)
// If the user enters a negative number, the loop terminates

#include <stdio.h>

int main() {
   int i;
   double number, sum = 0.0;

   for (i = 1; i <= 10; ++i) {
      printf("Enter n%d: ", i);
      scanf("%lf", &number);

      // if the user enters a negative number, break the loop
      if (number < 0.0) {
         break;
      }

      sum += number; // sum = sum + number;
   }

   printf("Sum = %.2lf", sum);

   return 0;
}
```

Output

```
Enter n1: 2.4
Enter n2: 4.5
Enter n3: 3.4
Enter n4: -3
Sum = 10.30
```

This program calculates the sum of a maximum of 10 numbers. Why a maximum of 10 numbers? It's because if the user enters a negative number, the `break` statement is executed. This will end the `for` loop, and the `sum` is displayed.

In C, `break` is also used with the `switch` statement. This will be discussed in the next tutorial.

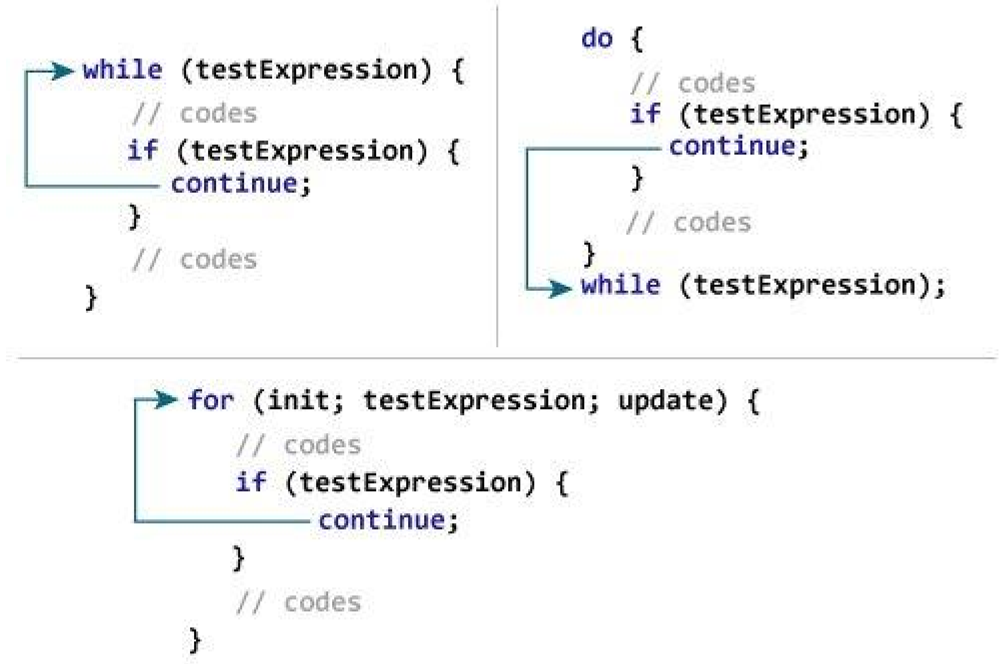
C continue

The `continue` statement skips the current iteration of the loop and continues with the next iteration. Its syntax is:

```
continue;
```

The `continue` statement is almost always used with the `if...else` statement.

How continue statement works?



```
while (testExpression) {
    // codes
    if (testExpression) {
      continue;
    }
    // codes
}
```

```
do {
    // codes
    if (testExpression) {
      continue;
    }
    // codes
}
while (testExpression);
```

```
for (init; testExpression; update) {
    // codes
    if (testExpression) {
      continue;
    }
    // codes
}
```

Working of Continue in C

Example 2: continue statement

```
// Program to calculate the sum of numbers (10 numbers max)
// If the user enters a negative number, it's not added to the result

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

```
    double number, sum = 0.0;

    for (i = 1; i <= 10; ++i) {
        printf("Enter a n%d: ", i);
        scanf("%lf", &number);

        if (number < 0.0) {
            continue;
        }

        sum += number; // sum = sum + number;
    }

    printf("Sum = %.2lf", sum);

    return 0;
}
```

Output

```
Enter n1: 1.1
Enter n2: 2.2
Enter n3: 5.5
Enter n4: 4.4
Enter n5: -3.4
Enter n6: -45.5
Enter n7: 34.5
Enter n8: -4.2
Enter n9: -1000
Enter n10: 12
Sum = 59.70
```

In this program, when the user enters a positive number, the sum is calculated using `sum += number;` statement.

When the user enters a negative number, the `continue` statement is executed and it skips the negative number from the calculation.

C switch Statement

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

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

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

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.

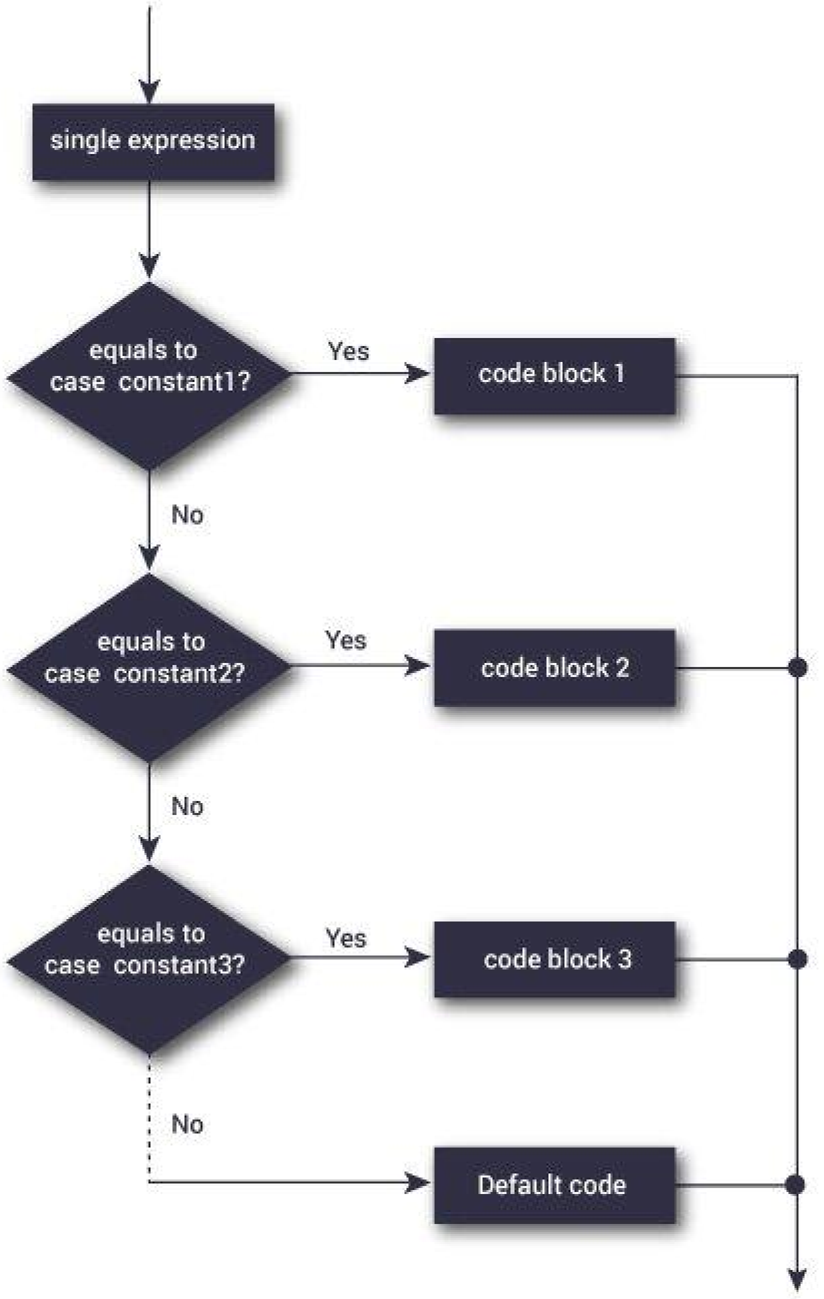
- If there is no match, the default statements are executed.

Notes:

- If we do not use the `break` statement, all statements after the matching label are also executed.
- The `default` clause inside the `switch` statement is optional.

---

switch Statement Flowchart



switch Statement

Flowchart

Example: Simple Calculator

```
// Program to create a simple calculator
#include <stdio.h>

int main() {
    char operation;
    double n1, n2;

    printf("Enter an operator (+, -, *, /): ");
    scanf("%c", &operation);
    printf("Enter two operands: ");
    scanf("%lf %lf",&n1, &n2);

    switch(operation)
    {
        case '+':
            printf("%.1lf + %.1lf = %.1lf",n1, n2, n1+n2);
            break;

        case '-':
            printf("%.1lf - %.1lf = %.1lf",n1, n2, n1-n2);
            break;

        case '*':
            printf("%.1lf * %.1lf = %.1lf",n1, n2, n1*n2);
            break;

        case '/':
            printf("%.1lf / %.1lf = %.1lf",n1, n2, n1/n2);
            break;

        // operator doesn't match any case constant +, -, *, /
        default:
            printf("Error! operator is not correct");
    }

    return 0;
}
```

Output

```
Enter an operator (+, -, *, /): -
Enter two operands: 32.5
12.4
32.5 - 12.4 = 20.1
```

The `-` operator entered by the user is stored in the `operation` variable. And, two operands `32.5` and `12.4` are stored in variables `n1` and `n2` respectively.

Since the `operation` is `-`, the control of the program jumps to

```
printf("%.1lf - %.1lf = %.1lf", n1, n2, n1-n2);
```

Finally, the break statement terminates the `switch` statement.

C Control Flow Examples

Check whether a number is even or odd

Check whether a character is a vowel or consonant

Find the largest number among three numbers

Find all roots of a quadratic equation

Check Whether the Entered Year is Leap Year or not

Check Whether a Number is Positive or Negative or Zero.

Checker whether a character is an alphabet or not

Find the sum of natural numbers

Find factorial of a number

Generate multiplication table

Display Fibonacci series

Find HCF of two numbers

Find LCM of two numbers

Count number of digits of an integer

Reverse a number

Calculate the power of a number

Check whether a number is a palindrome or not

Check whether an integer is prime or Not

Display prime numbers between two intervals

Check Armstrong number

Display Armstrong numbers between two intervals

Display factors of a number

Print pyramids and triangles

Create a simple calculator

C Functions

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 and reuse.

Types of function

There are two types of function in C programming:

- Standard library functions
- 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 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();

    ... .. ...
    ... .. ...
}
```

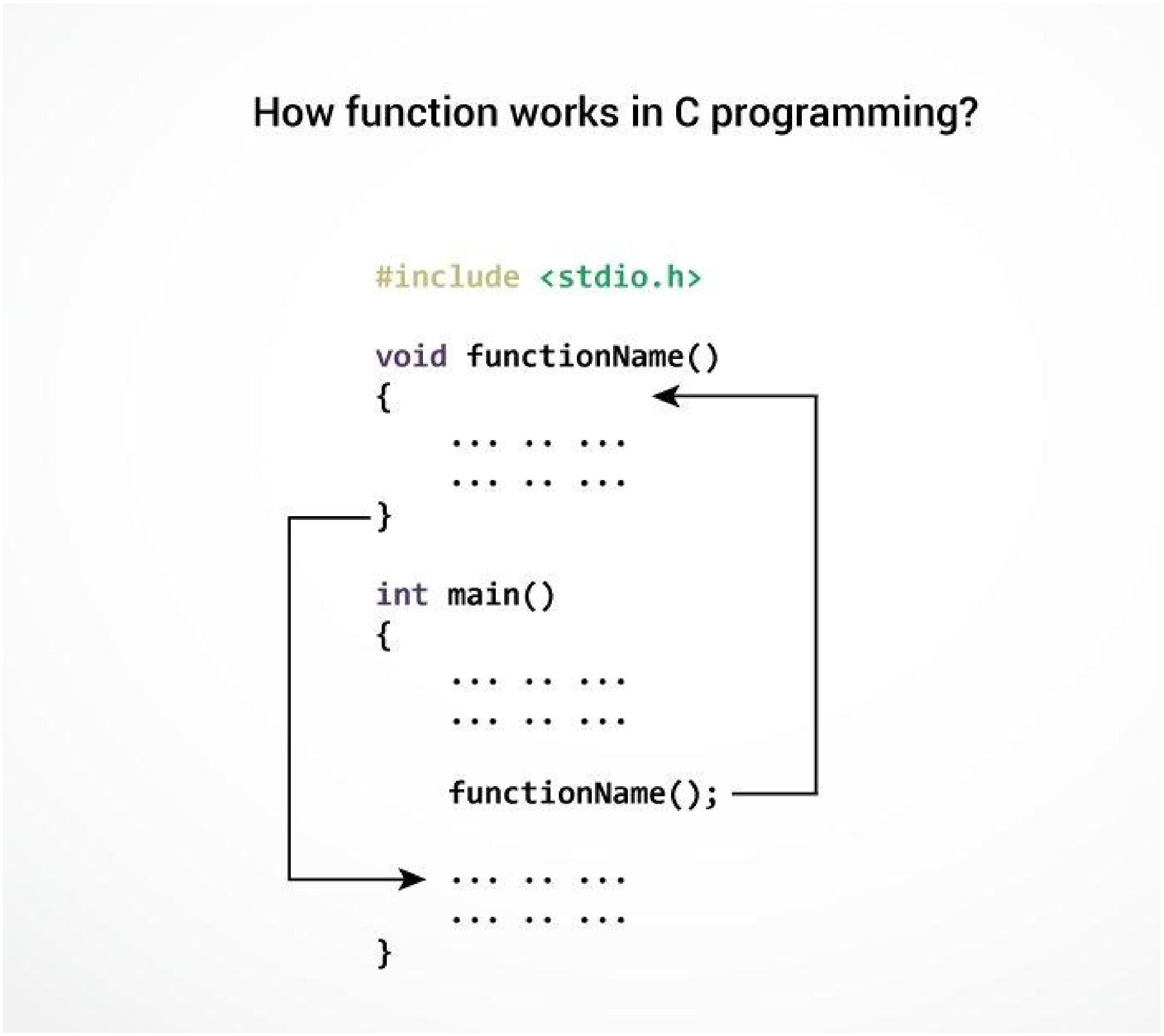
The execution of a C program begins from the `main()` function.

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.



```
#include <stdio.h>

void functionName()
{
    ... .. ...
    ... .. ...
}

int main()
{
    ... .. ...
    ... .. ...

    functionName();

    ... .. ...
    ... .. ...
}
```

Working of C Function

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
- Types of 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()`.

```c
#include <stdio.h>
int addNumbers(int a, int b);         // function prototype

int main()
{
    int n1,n2,sum;

    printf("Enters two numbers: ");
    scanf("%d %d",&n1,&n2);

    sum = addNumbers(n1, n2);        // function call
    printf("sum = %d",sum);

    return 0;
}

int addNumbers(int a, int b)         // function definition
{
    int result;
    result = a+b;
    return result;                  // return statement
}
```

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, ...)
```

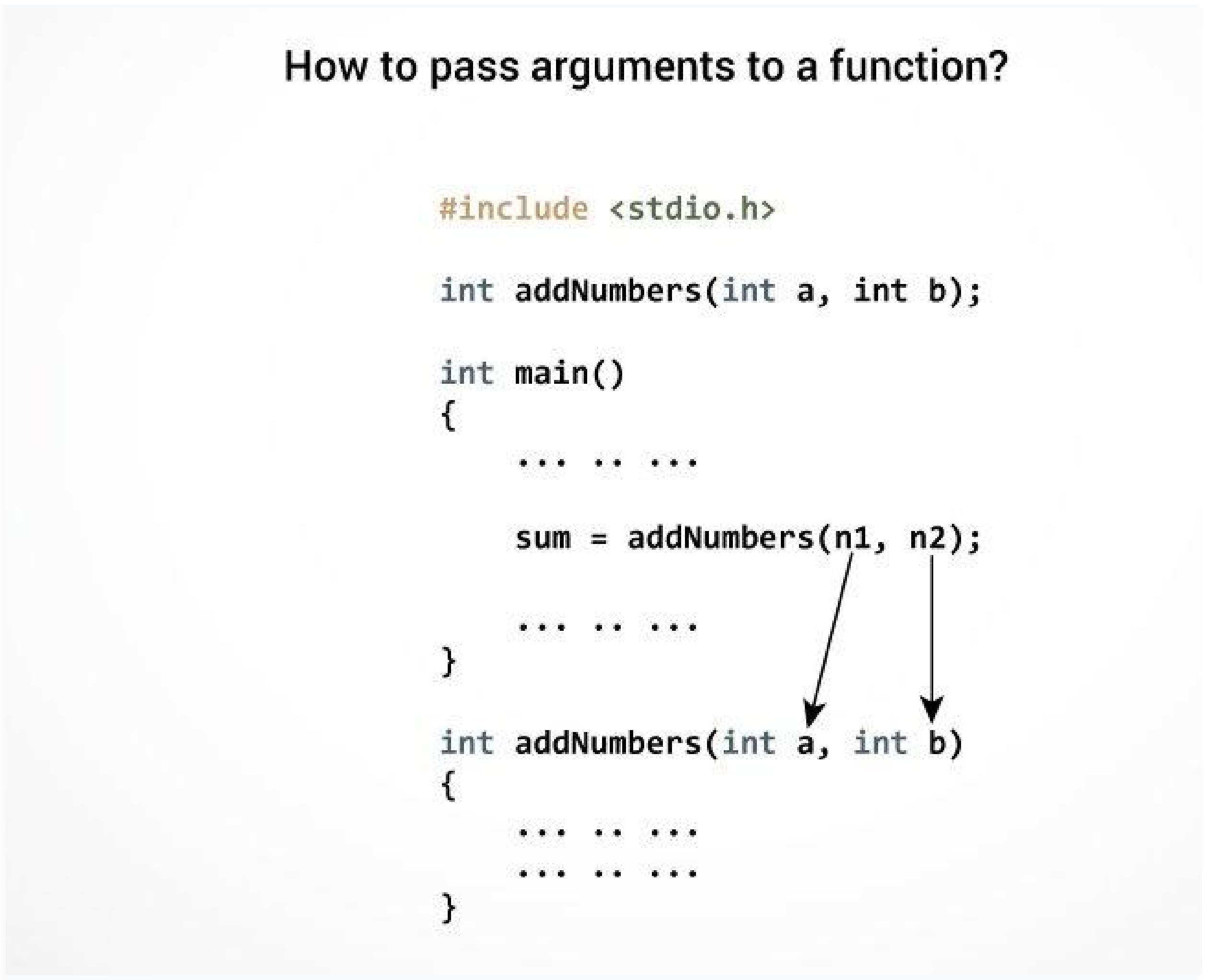
```
{
    //body of the function
}
```

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.


How to pass arguments to a function?

```
#include <stdio.h>

int addNumbers(int a, int b);

int main()
{
    ... .. ...

    sum = addNumbers(n1, n2);

    ... .. ...
}

int addNumbers(int a, int b)
{
    ... .. ...
    ... .. ...
}
```

Passing Argument to Function

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

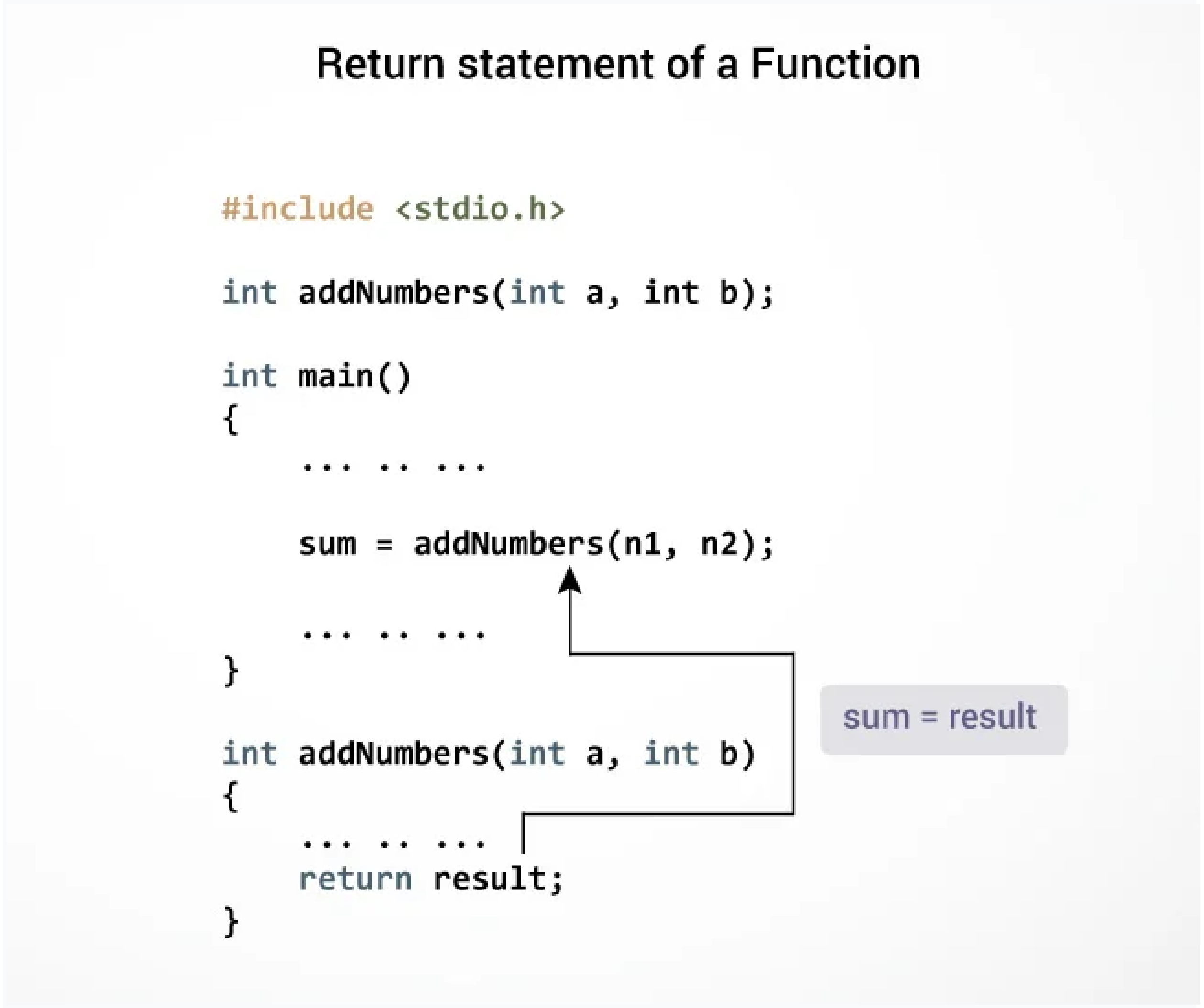
If `n1` is of char type, `a` also should be of char type. If `n2` is of float type, variable `b` also should be of float type.

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.



```
#include <stdio.h>

int addNumbers(int a, int b);

int main()
{
    ... .. ...

    sum = addNumbers(n1, n2);

    ... .. ...
}

int addNumbers(int a, int b)
{
    ... .. ...
    return result;
}
```

Return Statement of Function

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 Argument Passed and No Return Value

```
#include <stdio.h>

void checkPrimeNumber();

int main() {
  checkPrimeNumber();    // argument is not passed
  return 0;
}
```

```
// return type is void meaning doesn't return any value
void checkPrimeNumber() {
  int n, i, flag = 0;

  printf("Enter a positive integer: ");
  scanf("%d",&n);

  // 0 and 1 are not prime numbers
  if (n == 0 || n == 1)
    flag = 1;

  for(i = 2; i <= n/2; ++i) {
    if(n%i == 0) {
      flag = 1;
      break;
    }
  }

  if (flag == 1)
    printf("%d is not a prime number.", n);
  else
    printf("%d is a prime number.", n);
}
```

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();` 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 Returns a Value

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

int main() {
  int n, i, flag = 0;

  // no argument is passed
  n = getInteger();

  // 0 and 1 are not prime numbers
  if (n == 0 || n == 1)
    flag = 1;

  for(i = 2; i <= n/2; ++i) {
    if(n%i == 0){
      flag = 1;
      break;
    }
  }

  if (flag == 1)
    printf("%d is not a prime number.", n);
  else
    printf("%d is a prime number.", n);

  return 0;
}

// returns integer entered by the user
int getInteger() {
  int n;

  printf("Enter a positive integer: ");
  scanf("%d",&n);

  return n;
}
```

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

```
#include <stdio.h>
void checkPrimeAndDisplay(int n);

int main() {

  int n;

  printf("Enter a positive integer: ");
  scanf("%d",&n);

  // n is passed to the function
  checkPrimeAndDisplay(n);

  return 0;
}

// return type is void meaning doesn't return any value
void checkPrimeAndDisplay(int n) {
  int i, flag = 0;

  // 0 and 1 are not prime numbers
  if (n == 0 || n == 1)
    flag = 1;

  for(i = 2; i <= n/2; ++i) {
    if(n%i == 0){
      flag = 1;
      break;
    }
  }

  if(flag == 1)
    printf("%d is not a prime number.",n);
  else
    printf("%d is a prime number.", n);
}
```

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 Returns a Value

```
#include <stdio.h>
int checkPrimeNumber(int n);

int main() {

  int n, flag;

  printf("Enter a positive integer: ");
  scanf("%d",&n);

  // n is passed to the checkPrimeNumber() function
  // the returned value is assigned to the flag variable
  flag = checkPrimeNumber(n);

  if(flag == 1)
    printf("%d is not a prime number",n);
  else
    printf("%d is a prime number",n);

  return 0;
}

// int is returned from the function
int checkPrimeNumber(int n) {

  // 0 and 1 are not prime numbers
  if (n == 0 || n == 1)
    return 1;

  int i;

  for(i=2; i <= n/2; ++i) {
    if(n%i == 0)
      return 1;
  }

  return 0;
}
```

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 an argument and returning a value from the function (example 4) is better.

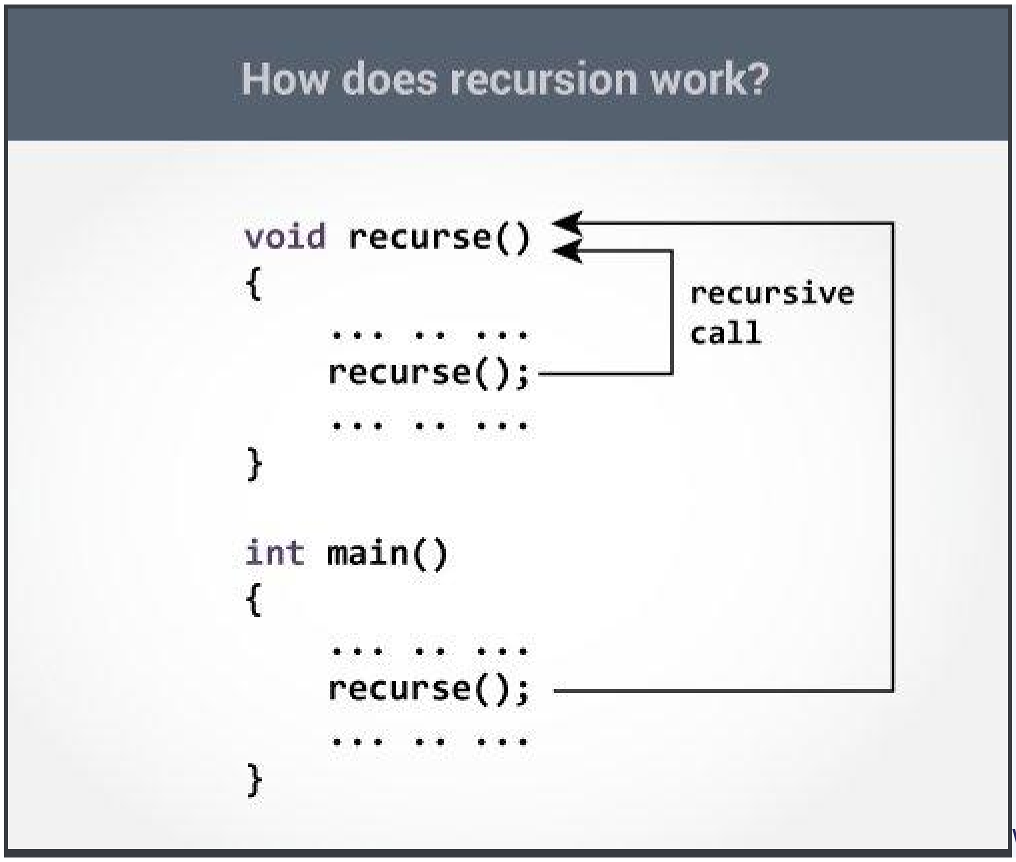
A function should perform a specific task. The `checkPrimeNumber()` function doesn't take input from the user nor it displays the appropriate message. It only checks whether a number is prime or not.

C Recursion

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();
    ... .. ...
}
```



```
void recurse()
{
    ... .. ...
    recurse();
    ... .. ...
}

int main()
{
    ... .. ...
    recurse();
    ... .. ...
}
```

Working of Recursion

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

To prevent infinite recursion, 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

```
#include <stdio.h>
int sum(int n);

int main() {
    int number, result;

    printf("Enter a positive integer: ");
    scanf("%d", &number);

    result = sum(number);
    printf("sum = %d", result);
    return 0;
}

int sum(int n) {
    if (n != 0)
        // sum() function calls itself
        return n + sum(n-1);
    else
        return n;
}
```

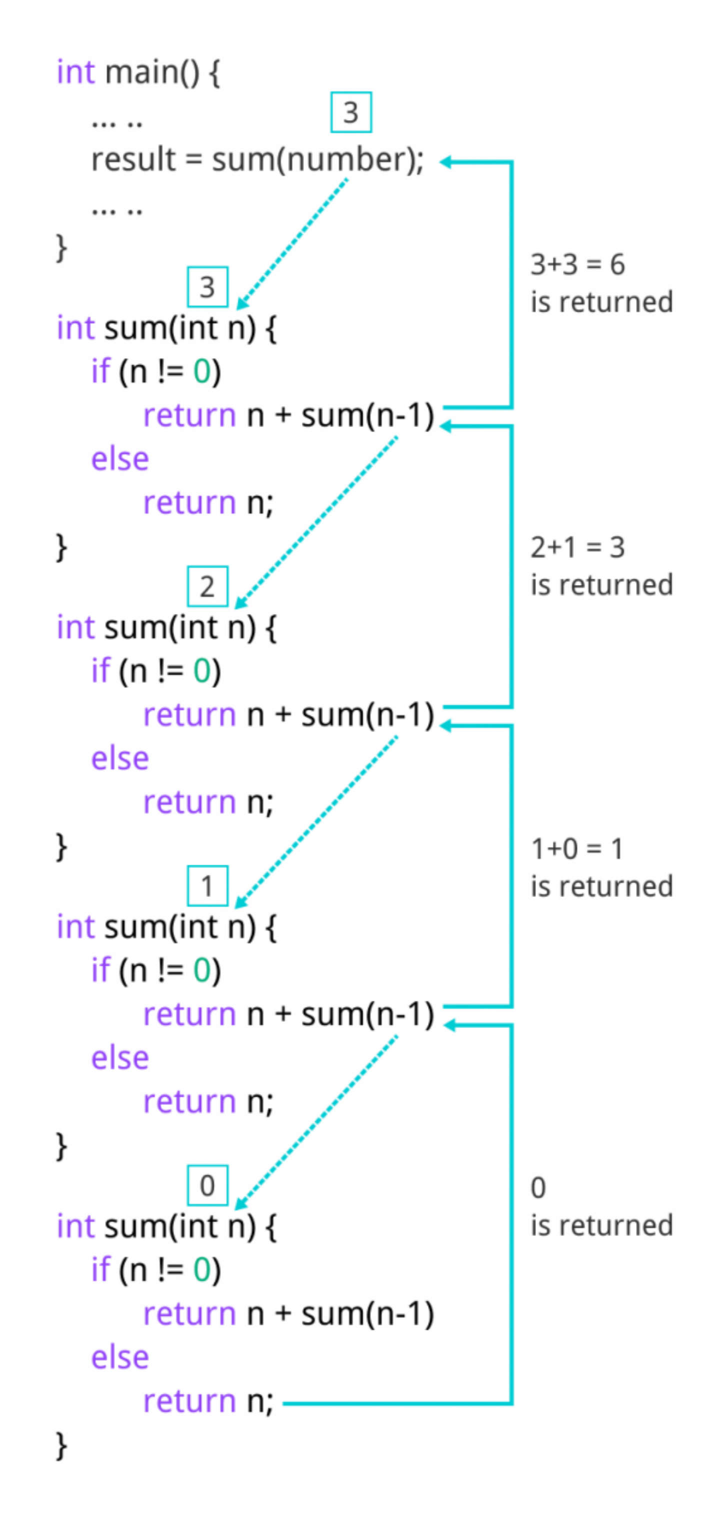
Output

```
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.



```
int main() {
    ... ..
    result = sum(number);
    ... ..
}

int sum(int n) {
    if (n != 0)
        return n + sum(n-1)
    else
        return n;
}
```

Sum of Natural Numbers

Advantages and Disadvantages of Recursion

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

C Storage Class

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.

```
#include <stdio.h>

int main(void) {

  for (int i = 0; i < 5; ++i) {
     printf("C programming");
  }

 // Error: i is not declared at this point
```

```
  printf("%d", i);
  return 0;
}
```

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.

```
int main() {
    int n1; // n1 is a local variable to main()
}

void func() {
   int n2;  // n2 is a local variable to func()
}
```

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

```
#include <stdio.h>
void display();
```

```
int n = 5;  // global variable

int main()
{
    ++n;
    display();
    return 0;
}

void display()
{
    ++n;
    printf("n = %d", n);
}
```

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

```
#include <stdio.h>
void display();

int main()
{
    display();
    display();
}
void display()
{
    static int c = 1;
    c += 5;
    printf("%d  ",c);
}
```

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 Function Examples

Display all prime numbers between two Intervals

Check prime and Armstrong number by making functions

Check whether a number can be expressed as the sum of two prime numbers

Find the sum of natural numbers using recursion

Calculate the factorial of a number using recursion

Find G.C.D using recursion

Reverse a sentence using recursion

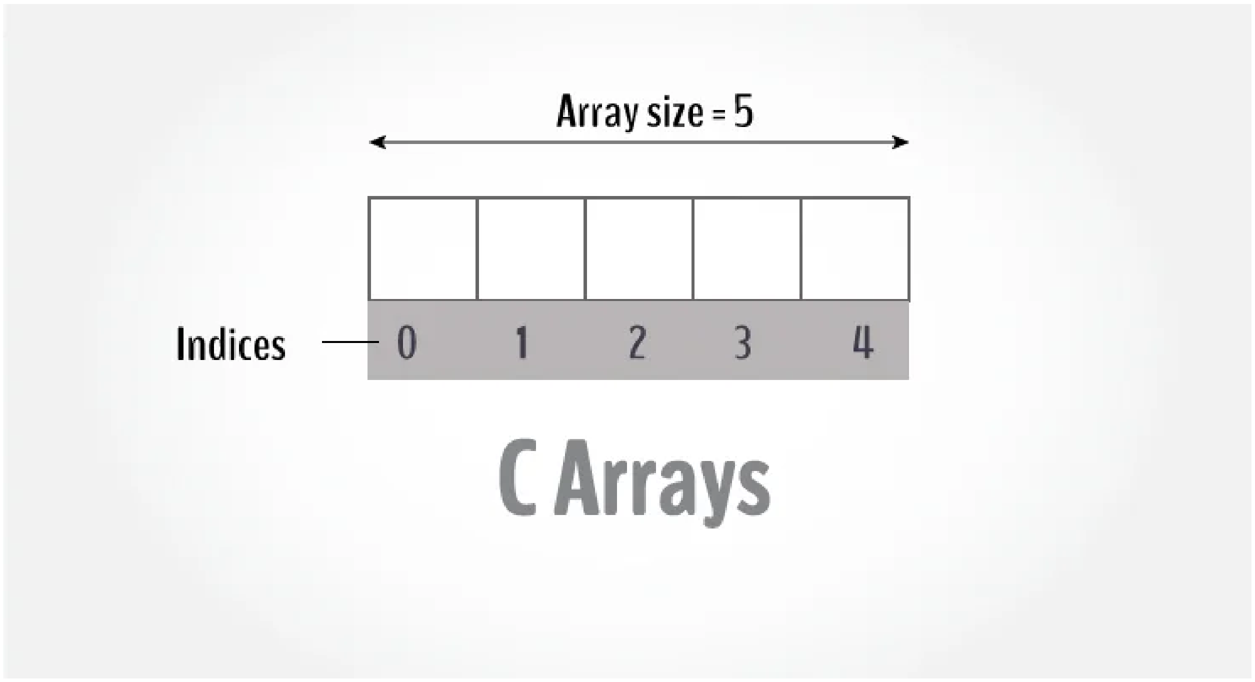
Calculate the power of a number using recursion

Convert a binary number to decimal and vice-versa

Convert an octal Number to decimal and vice-versa

Convert a binary number to octal and vice-versa

C Arrays



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.

```
int data[100];
```

How to declare an array?

```
dataType arrayName[arraySize];
```

For example,

```
float mark[5];
```

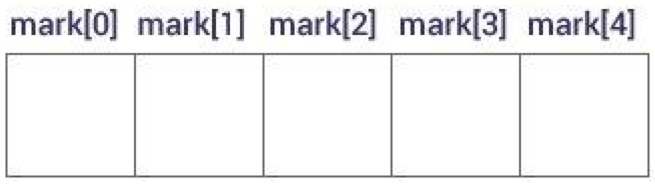
Here, we declared an array, `mark`, of floating-point type. And its size is 5. Meaning, it can hold 5 floating-point values.

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.



Declare an Array

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?

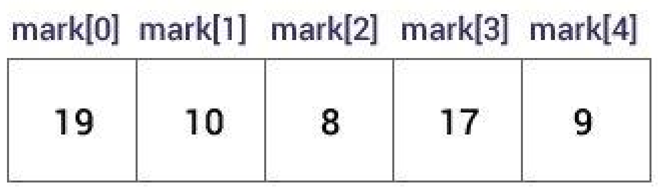
It is possible to initialize an array during declaration. For example,

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

You can also initialize an array like this.

```
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.



Initialize an Array

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

```
int mark[5] = {19, 10, 8, 17, 9}

// make the value of the third element to -1
mark[2] = -1;

// make the value of the fifth element to 0
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.

```
// take input and store it in the 3rd element
scanf("%d", &mark[2]);

// take input and store it in the ith element
scanf("%d", &mark[i-1]);
```

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

```
// print the first element of the array
printf("%d", mark[0]);

// print the third element of the array
printf("%d", mark[2]);

// print ith element of the array
printf("%d", mark[i-1]);
```

Example 1: Array Input/Output

```
// Program to take 5 values from the user and store them in an array
// Print the elements stored in the array
#include <stdio.h>

int main() {
  int values[5];

  printf("Enter 5 integers: ");

  // taking input and storing it in an array
  for(int i = 0; i < 5; ++i) {
     scanf("%d", &values[i]);
  }

  printf("Displaying integers: ");
  // printing elements of an array
  for(int i = 0; i < 5; ++i) {
     printf("%d\n", values[i]);
  }
  return 0;
}
```

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

```
// Program to find the average of n numbers using arrays

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

  int marks[10], i, n, sum = 0, average;

  printf("Enter number of elements: ");
  scanf("%d", &n);

  for(i=0; i < n; ++i) {
    printf("Enter number%d: ",i+1);
    scanf("%d", &marks[i]);
```

```
     // adding integers entered by the user to the sum variable
     sum += marks[i];
   }

   average = sum / n;
   printf("Average = %d", average);

   return 0;
}
```

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.

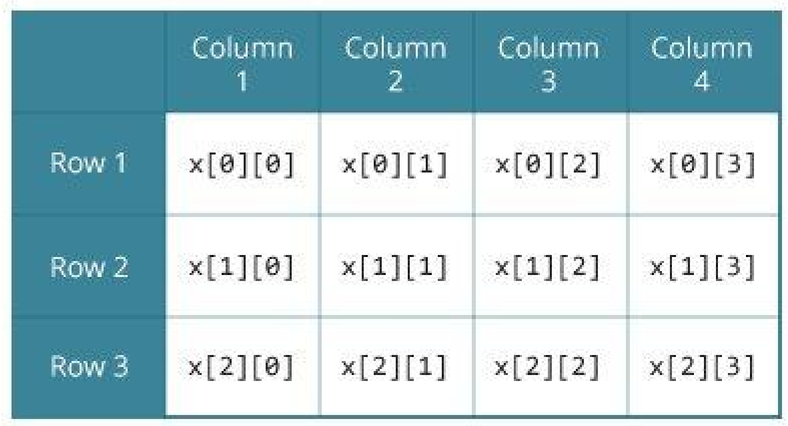
Hence, you should never access elements of an array outside of its bound.

C Multidimensional Arrays

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

```
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.



	Column 1	Column 2	Column 3	Column 4
Row 1	x[0][0]	x[0][1]	x[0][2]	x[0][3]
Row 2	x[1][0]	x[1][1]	x[1][2]	x[1][3]
Row 3	x[2][0]	x[2][1]	x[2][2]	x[2][3]

Two dimensional Array

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

```
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

```
// Different ways to initialize two-dimensional array

int c[2][3] = {{1, 3, 0}, {-1, 5, 9}};

int c[][3] = {{1, 3, 0}, {-1, 5, 9}};

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,

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

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

```
// C program to store temperature of two cities of a week and display it.
#include <stdio.h>
const int CITY = 2;
const int WEEK = 7;
int main()
{
  int temperature[CITY][WEEK];

  // Using nested loop to store values in a 2d array
  for (int i = 0; i < CITY; ++i)
  {
    for (int j = 0; j < WEEK; ++j)
    {
      printf("City %d, Day %d: ", i + 1, j + 1);
      scanf("%d", &temperature[i][j]);
    }
  }
  printf("\nDisplaying values: \n\n");

  // Using nested loop to display vlues of a 2d array
  for (int i = 0; i < CITY; ++i)
  {
    for (int j = 0; j < WEEK; ++j)
    {
      printf("City %d, Day %d = %d\n", i + 1, j + 1, temperature[i][j]);
    }
  }
  return 0;
}
```

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

```c
// C program to find the sum of two matrices of order 2*2

#include <stdio.h>
int main()
{
  float a[2][2], b[2][2], result[2][2];

  // Taking input using nested for loop
  printf("Enter elements of 1st matrix\n");
  for (int i = 0; i < 2; ++i)
    for (int j = 0; j < 2; ++j)
    {
      printf("Enter a%d%d: ", i + 1, j + 1);
      scanf("%f", &a[i][j]);
    }

  // Taking input using nested for loop
  printf("Enter elements of 2nd matrix\n");
  for (int i = 0; i < 2; ++i)
    for (int j = 0; j < 2; ++j)
    {
      printf("Enter b%d%d: ", i + 1, j + 1);
      scanf("%f", &b[i][j]);
    }

  // adding corresponding elements of two arrays
  for (int i = 0; i < 2; ++i)
    for (int j = 0; j < 2; ++j)
    {
      result[i][j] = a[i][j] + b[i][j];
    }

  // Displaying the sum
  printf("\nSum Of Matrix:");

  for (int i = 0; i < 2; ++i)
    for (int j = 0; j < 2; ++j)
    {
      printf("%.1f\t", result[i][j]);

      if (j == 1)
        printf("\n");
    }
  return 0;
}
```

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

```c
// C Program to store and print 12 values entered by the user

#include <stdio.h>
int main()
{
  int test[2][3][2];

  printf("Enter 12 values: \n");

  for (int i = 0; i < 2; ++i)
  {
    for (int j = 0; j < 3; ++j)
    {
      for (int k = 0; k < 2; ++k)
      {
        scanf("%d", &test[i][j][k]);
      }
    }
  }

  // Printing values with proper index.

  printf("\nDisplaying values:\n");
  for (int i = 0; i < 2; ++i)
  {
    for (int j = 0; j < 3; ++j)
    {
      for (int k = 0; k < 2; ++k)
```

```
        {
          printf("test[%d][%d][%d] = %d\n", i, j, k, test[i][j][k]);
        }
      }
    }

  return 0;
}
```

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
```

Pass arrays to a function in C

In this tutorial, you'll learn to pass arrays (both one-dimensional and multidimensional arrays) to a function in C programming with the help of examples.

In C programming, you can pass an entire array to functions. Before we learn that, let's see how you can pass individual elements of an array to functions.

Pass Individual Array Elements

Passing array elements to a function is similar to passing variables to a function.

Example 1: Pass Individual Array Elements

```
#include <stdio.h>
void display(int age1, int age2) {
  printf("%d\n", age1);
  printf("%d\n", age2);
}

int main() {
  int ageArray[] = {2, 8, 4, 12};

  // pass second and third elements to display()
  display(ageArray[1], ageArray[2]);
  return 0;
}
```

Output

```
8
4
```

Here, we have passed array parameters to the `display()` function in the same way we pass variables to a function.

```
// pass second and third elements to display()
display(ageArray[1], ageArray[2]);
```

We can see this in the function definition, where the function parameters are individual variables:

```
void display(int age1, int age2) {
  // code
}
```

Example 2: Pass Arrays to Functions

```
// Program to calculate the sum of array elements by passing to a function

#include <stdio.h>
float calculateSum(float num[]);

int main() {
  float result, num[] = {23.4, 55, 22.6, 3, 40.5, 18};

  // num array is passed to calculateSum()
  result = calculateSum(num);
  printf("Result = %.2f", result);
  return 0;
}

float calculateSum(float num[]) {
  float sum = 0.0;

  for (int i = 0; i < 6; ++i) {
    sum += num[i];
  }

  return sum;
}
```

Output

```
Result = 162.50
```

To pass an entire array to a function, only the name of the array is passed as an argument.

```
result = calculateSum(num);
```

However, notice the use of `[]` in the function definition.

```
float calculateSum(float num[]) {
... ..
}
```

This informs the compiler that you are passing a one-dimensional array to the function.

Pass Multidimensional Arrays to a Function

To pass multidimensional arrays to a function, only the name of the array is passed to the function (similar to one-dimensional arrays).

Example 3: Pass two-dimensional arrays

```
#include <stdio.h>
void displayNumbers(int num[2][2]);

int main() {
  int num[2][2];
  printf("Enter 4 numbers:\n");
  for (int i = 0; i < 2; ++i) {
    for (int j = 0; j < 2; ++j) {
      scanf("%d", &num[i][j]);
    }
  }

  // pass multi-dimensional array to a function
  displayNumbers(num);
  return 0;
}

void displayNumbers(int num[2][2]) {
  printf("Displaying:\n");
  for (int i = 0; i < 2; ++i) {
    for (int j = 0; j < 2; ++j) {
      printf("%d\n", num[i][j]);
    }
  }
}
```

Output

```
Enter 4 numbers:
2
3
4
5
Displaying:
2
3
4
5
```

Notice the parameter `int num[2][2]` in the function prototype and function definition:

```
// function prototype
void displayNumbers(int num[2][2]);
```

This signifies that the function takes a two-dimensional array as an argument. We can also pass arrays with more than 2 dimensions as a function argument.

When passing two-dimensional arrays, it is not mandatory to specify the number of rows in the array. However, the number of columns should always be specified.

For example,

```
void displayNumbers(int num[][2]) {
   // code
}
```

Note: In C programming, you can pass arrays to functions, however, you cannot return arrays from functions.

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.

```
scanf("%d", &var);
```

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

```
#include <stdio.h>
int main()
{
  int var = 5;
  printf("var: %d\n", var);

  // Notice the use of & before var
  printf("address of var: %p", &var);
  return 0;
}
```

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.

```
int* p;
```

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

You can also declare pointers in these ways.

```
int *p1;
int * p2;
```

Let's take another example of declaring pointers.

```
int* p1, p2;
```

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

Assigning addresses to Pointers

Let's take an example.

```
int* pc, c;
c = 5;
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:

```
int* pc, c;
c = 5;
pc = &c;
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.

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

```
int* pc, c;
c = 5;
pc = &c;
*pc = 1;
printf("%d", *pc);  // Ouptut: 1
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.

```
int* pc, c, d;
c = 5;
d = -15;

pc = &c; printf("%d", *pc); // Output: 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.

```
#include <stdio.h>
int main()
{
   int* pc, c;

   c = 22;
   printf("Address of c: %p\n", &c);
   printf("Value of c: %d\n\n", c);  // 22

   pc = &c;
   printf("Address of pointer pc: %p\n", pc);
   printf("Content of pointer pc: %d\n\n", *pc); // 22

   c = 11;
   printf("Address of pointer pc: %p\n", pc);
   printf("Content of pointer pc: %d\n\n", *pc); // 11

   *pc = 2;
   printf("Address of c: %p\n", &c);
   printf("Value of c: %d\n\n", c); // 2
   return 0;
}
```

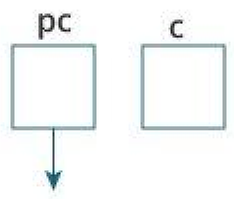
Output

```
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
```

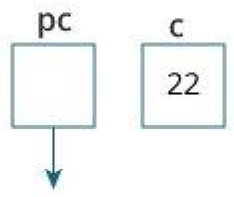
Explanation of the program

1. `int* pc, c;`

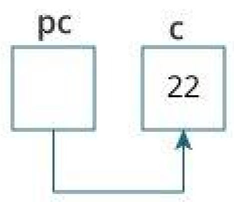
   Here, a pointer `pc` and a normal variable `c`, both of type `int`, is created. Since `pc` and `c` are not initialized at initially, pointer `pc` points to either no address or a random address. And, variable `c` has an address but contains random garbage value.

2. `c = 22;`

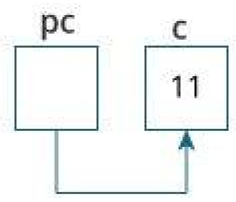
   This assigns 22 to the variable `c`. That is, 22 is stored in the memory location of variable `c`.

3. `pc = &c;`



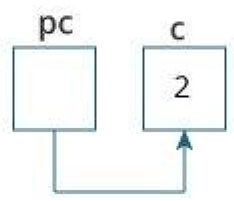
This assigns the address of variable `c` to the pointer `pc`.

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,

```
int c, *pc;
```

```
// pc is address but c is not
pc = c;   // Error

// &c is address but *pc is not
*pc = &c;   // Error

// both &c and pc are addresses
pc = &c;   // Not an error

// both c and *pc are values
*pc = c;   // Not an error
```

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

```
#include <stdio.h>
int main() {
   int c = 5;
   int *p = &c;

   printf("%d", *p);  // 5
   return 0;
}
```

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

It's because

```
int *p = &c;
```

is equivalent to

```
int *p:
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:

```
int* p = &c;
```

Relationship Between Arrays and Pointers

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

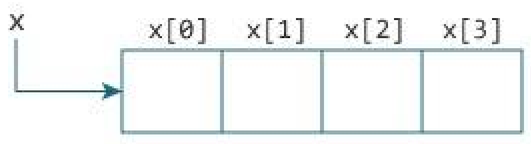
```
#include <stdio.h>
int main() {
   int x[4];
   int i;

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

   printf("Address of array x: %p", x);

   return 0;
}
```

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.



Relation between Arrays and Pointers

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

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

  int i, x[6], sum = 0;

  printf("Enter 6 numbers: ");

  for(i = 0; i < 6; ++i) {
  // Equivalent to scanf("%d", &x[i]);
      scanf("%d", x+i);

  // Equivalent to sum += x[i]
      sum += *(x+i);
  }

  printf("Sum = %d", sum);

  return 0;
}
```

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?

Example 2: Arrays and Pointers

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

  int x[5] = {1, 2, 3, 4, 5};
  int* ptr;

  // ptr is assigned the address of the third element
  ptr = &x[2];

  printf("*ptr = %d \n", *ptr);   // 3
  printf("*(ptr+1) = %d \n", *(ptr+1)); // 4
  printf("*(ptr-1) = %d", *(ptr-1));  // 2

  return 0;
}
```

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 Pass Addresses and Pointers

In this tutorial, you'll learn to pass addresses and pointers as arguments to functions with the help of examples.

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: Pass Addresses to Functions

```
#include <stdio.h>
void swap(int *n1, int *n2);

int main()
{
    int num1 = 5, num2 = 10;
    // address of num1 and num2 is passed
    swap( &num1, &num2);

    printf("num1 = %d\n", num1);
    printf("num2 = %d", num2);
    return 0;
}

void swap(int* n1, int* n2)
{
    int temp;
    temp = *n1;
    *n1 = *n2;
    *n2 = temp;
}
```

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.

```
void swap(int* n1, int* n2) {
    ... ..
}
```

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`.

Example 2: Passing Pointers to Functions

```
#include <stdio.h>

void addOne(int* ptr) {
  (*ptr)++; // adding 1 to *ptr
}

int main()
{
  int* p, i = 10;
  p = &i;
  addOne(p);

  printf("%d", *p); // 11
  return 0;
}
```

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 Dynamic Memory Allocation

In this tutorial, you'll learn to dynamically allocate memory in your C program using standard library functions: malloc(), calloc(), free() and realloc().

As you know, an array is a collection of a fixed number of values. Once the size of an array is declared, you cannot change it.

Sometimes the size of the array you declared may be insufficient. To solve this issue, you can allocate memory manually during run-time. This is known as dynamic memory allocation in C programming.

To allocate memory dynamically, library functions are `malloc()`, `calloc()`, `realloc()` and `free()` are used. These functions are defined in the `<stdlib.h>` header file.

C malloc()

The name "malloc" stands for memory allocation.

The `malloc()` function reserves a block of memory of the specified number of bytes. And, it returns a pointer of `void` which can be casted into pointers of any form.

Syntax of malloc()

```
ptr = (castType*) malloc(size);
```

Example

```
ptr = (float*) malloc(100 * sizeof(float));
```

The above statement allocates 400 bytes of memory. It's because the size of `float` is 4 bytes. And, the pointer `ptr` holds the address of the first byte in the allocated memory.

The expression results in a `NULL` pointer if the memory cannot be allocated.

C calloc()

The name "calloc" stands for contiguous allocation.

The `malloc()` function allocates memory and leaves the memory uninitialized, whereas the `calloc()` function allocates memory and initializes all bits to zero.

Syntax of calloc()

```
ptr = (castType*)calloc(n, size);
```

Example:

```
ptr = (float*) calloc(25, sizeof(float));
```

The above statement allocates contiguous space in memory for 25 elements of type `float`.

C free()

Dynamically allocated memory created with either `calloc()` or `malloc()` doesn't get freed on their own. You must explicitly use `free()` to release the space.

Syntax of free()

```
free(ptr);
```

This statement frees the space allocated in the memory pointed by `ptr`.

Example 1: malloc() and free()

```
// Program to calculate the sum of n numbers entered by the user

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

int main() {
  int n, i, *ptr, sum = 0;

  printf("Enter number of elements: ");
  scanf("%d", &n);

  ptr = (int*) malloc(n * sizeof(int));

  // if memory cannot be allocated
  if(ptr == NULL) {
    printf("Error! memory not allocated.");
    exit(0);
  }

  printf("Enter elements: ");
  for(i = 0; i < n; ++i) {
    scanf("%d", ptr + i);
    sum += *(ptr + i);
  }

  printf("Sum = %d", sum);

  // deallocating the memory
  free(ptr);

  return 0;
}
```

Output

```
Enter number of elements: 3
Enter elements: 100
20
36
Sum = 156
```

Here, we have dynamically allocated the memory for `n` number of `int`.

Example 2: calloc() and free()

```
// Program to calculate the sum of n numbers entered by the user

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

int main() {
  int n, i, *ptr, sum = 0;
  printf("Enter number of elements: ");
  scanf("%d", &n);

  ptr = (int*) calloc(n, sizeof(int));
  if(ptr == NULL) {
    printf("Error! memory not allocated.");
    exit(0);
  }

  printf("Enter elements: ");
  for(i = 0; i < n; ++i) {
    scanf("%d", ptr + i);
    sum += *(ptr + i);
  }

  printf("Sum = %d", sum);
  free(ptr);
  return 0;
}
```

Output

```
Enter number of elements: 3
Enter elements: 100
20
36
Sum = 156
```

C realloc()

If the dynamically allocated memory is insufficient or more than required, you can change the size of previously allocated memory using the `realloc()` function.

Syntax of realloc()

```
ptr = realloc(ptr, x);
```

Here, `ptr` is reallocated with a new size `x`.

Example 3: realloc()

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

int main() {
  int *ptr, i , n1, n2;
  printf("Enter size: ");
  scanf("%d", &n1);

  ptr = (int*) malloc(n1 * sizeof(int));

  printf("Addresses of previously allocated memory:\n");
  for(i = 0; i < n1; ++i)
```

```
    printf("%pc\n",ptr + i);

  printf("\nEnter the new size: ");
  scanf("%d", &n2);

  // rellocating the memory
  ptr = realloc(ptr, n2 * sizeof(int));

  printf("Addresses of newly allocated memory:\n");
  for(i = 0; i < n2; ++i)
    printf("%pc\n", ptr + i);

  free(ptr);

  return 0;
}
```

Output

```
Enter size: 2
Addresses of previously allocated memory:
26855472
26855476

Enter the new size: 4
Addresses of newly allocated memory:
26855472
26855476
26855480
26855484
```

Array and Pointer Examples

Calculate the average of array elements

Find the largest element of an array

Calculate standard deviation

Add two matrices

Multiply two matrices

Find transpose of a matrix

Multiply two matrices

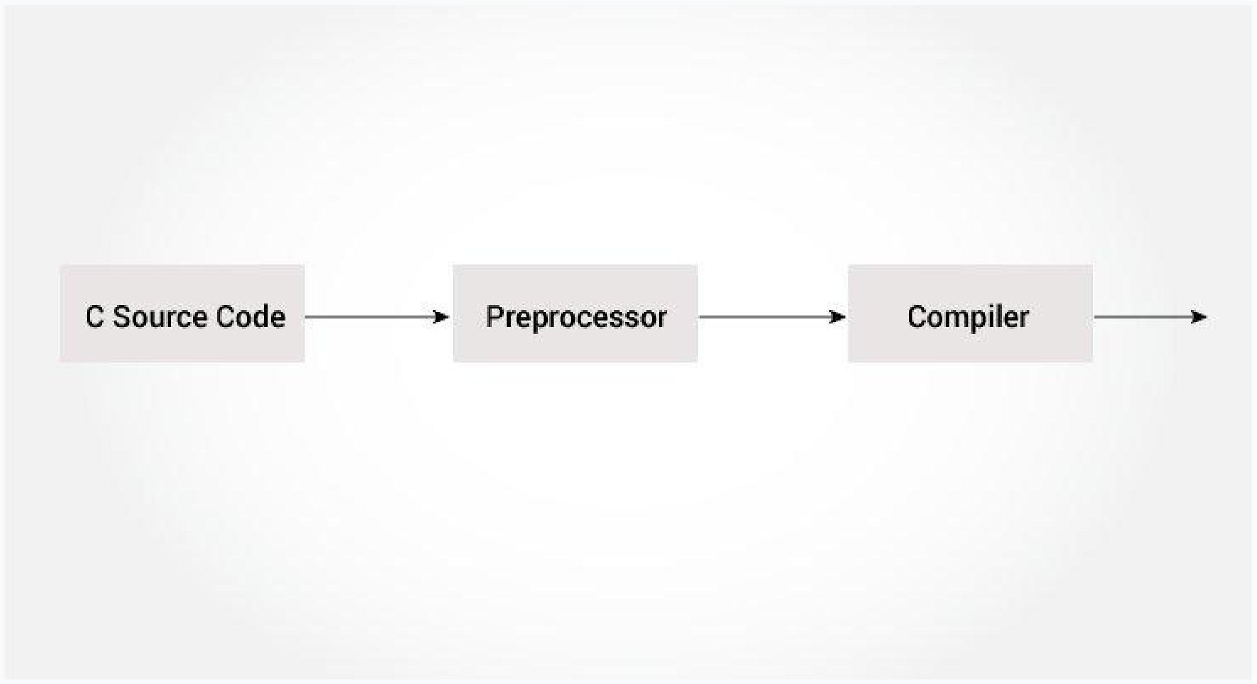
Access elements of an array using pointers

Swap numbers in the cyclic order using call by reference

Find the largest number (Dynamic memory allocation is used)

C Preprocessor and Macros

In this tutorial, you will be introduced to c preprocessors, and you will learn to use #include, #define and conditional compilation with the help of examples.



Working of C Preprocessor

The C preprocessor is a macro preprocessor (allows you to define macros) that transforms your program before it is compiled. These transformations can be the inclusion of header file, macro expansions etc.

All preprocessing directives begin with a `#` symbol. For example,

```
#define PI 3.14
```

Some of the common uses of C preprocessor are:

Including Header Files: #include

The `#include` preprocessor is used to include header files to C programs. For example,

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

Here, `stdio.h` is a header file. The `#include` preprocessor directive replaces the above line with the contents of `stdio.h` header file.

That's the reason why you need to use `#include <stdio.h>` before you can use functions like `scanf()` and `printf()`.

You can also create your own header file containing function declaration and include it in your program using this preprocessor directive.

```
#include "my_header.h"
```

Visit this page to learn more about using header files.

Macros using #define

A macro is a fragment of code that is given a name. You can define a macro in C using the `#define` preprocessor directive.

Here's an example.

```
#define c 299792458  // speed of light
```

Here, when we use `c` in our program, it is replaced with `299792458`.

Example 1: #define preprocessor

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

```
#define PI 3.1415

int main()
{
    float radius, area;
    printf("Enter the radius: ");
    scanf("%f", &radius);

    // Notice, the use of PI
    area = PI*radius*radius;

    printf("Area=%.2f",area);
    return 0;
}
```

Function like Macros

You can also define macros that work in a similar way like a function call. This is known as function-like macros. For example,

```
#define circleArea(r) (3.1415*(r)*(r))
```

Every time the program encounters `circleArea(argument)`, it is replaced by `(3.1415*(argument)*(argument))`.

Suppose, we passed 5 as an argument then, it expands as below:

```
circleArea(5) expands to (3.1415*5*5)
```

Example 2: Using #define preprocessor

```
#include <stdio.h>
#define PI 3.1415
#define circleArea(r) (PI*r*r)

int main() {
    float radius, area;

    printf("Enter the radius: ");
    scanf("%f", &radius);
    area = circleArea(radius);
    printf("Area = %.2f", area);

    return 0;
}
```

Visit this page to learn more about macros and #define preprocessor.

Conditional Compilation

In C programming, you can instruct preprocessor whether to include a block of code or not. To do so, conditional directives can be used.

It's similar to a `if` statement with one major difference.

The `if` statement is tested during the execution time to check whether a block of code should be executed or not whereas, the conditionals are used to include (or skip) a block of code in your program before execution.

Uses of Conditional

- use different code depending on the machine, operating system
- compile same source file in two different programs
- to exclude certain code from the program but to keep it as reference for future purpose

How to use conditional?

To use conditional, `#ifdef`, `#if`, `#defined`, `#else` and `#elif` directives are used.

#ifdef Directive

```
#ifdef MACRO
   // conditional codes
#endif
```

Here, the conditional codes are included in the program only if `MACRO` is defined.

#if, #elif and #else Directive

```
#if expression
   // conditional codes
#endif
```

Here, `expression` is an expression of integer type (can be integers, characters, arithmetic expression, macros and so on).

The conditional codes are included in the program only if the `expression` is evaluated to a non-zero value.

The optional `#else` directive can be used with `#if` directive.

```
#if expression
   conditional codes if expression is non-zero
#else
   conditional if expression is 0
#endif
```

You can also add nested conditional to your `#if...#else` using `#elif`

```
#if expression
    // conditional codes if expression is non-zero
#elif expression1
    // conditional codes if expression is non-zero
#elif expression2
    // conditional codes if expression is non-zero
#else
    // conditional if all expressions are 0
#endif
```

#defined

The special operator `#defined` is used to test whether a certain macro is defined or not. It's often used with `#if` directive.

```
#if defined BUFFER_SIZE && BUFFER_SIZE >= 2048
  // codes
```

Predefined Macros

Here are some predefined macros in C programming.

Macro	Value
`__DATE__`	A string containing the current date
`__FILE__`	A string containing the file name
`__LINE__`	An integer representing the current line number

`__STDC__`	If follows ANSI standard C, then the value is a nonzero integer
`__TIME__`	A string containing the current date.

Example 3: Get current time using __TIME__

The following program outputs the current time using `__TIME__` macro.

```
#include <stdio.h>
int main()
{
   printf("Current time: %s",__TIME__);
}
```

Output

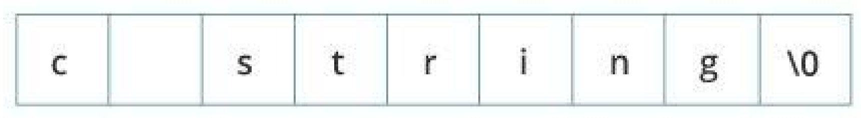
```
Current time: 19:54:39
```

C Programming Strings

In C programming, a string is a sequence of characters terminated with a null character `\0`. For example:

```
char c[] = "c string";
```

When the compiler encounters a sequence of characters enclosed in the double quotation marks, it appends a null character `\0` at the end by default.



Memory Diagram

How to declare a string?

Here's how you can declare strings:

```
char s[5];
```

s[0]
s[1]
s[2]
s[3]
s[4]
String Declaration in C

Here, we have declared a string of 5 characters.

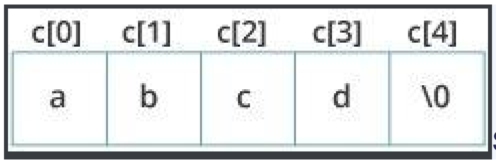
How to initialize strings?

You can initialize strings in a number of ways.

```
char c[] = "abcd";
char c[50] = "abcd";
```

```
char c[] = {'a', 'b', 'c', 'd', '\0'};
```

```
char c[5] = {'a', 'b', 'c', 'd', '\0'};
```



String Initialization in C

Let's take another example:

```
char c[5] = "abcde";
```

Here, we are trying to assign 6 characters (the last character is `'\0'`) to a `char` array having 5 characters. This is bad and you should never do this.

Assigning Values to Strings

Arrays and strings are second-class citizens in C; they do not support the assignment operator once it is declared. For example,

```
char c[100];
c = "C programming";  // Error! array type is not assignable.
```

Note: Use the strcpy() function to copy the string instead.

Read String from the user

You can use the `scanf()` function to read a string.

The `scanf()` function reads the sequence of characters until it encounters whitespace (space, newline, tab, etc.).

Example 1: scanf() to read a string

```
#include <stdio.h>
int main()
{
    char name[20];
    printf("Enter name: ");
    scanf("%s", name);
    printf("Your name is %s.", name);
    return 0;
}
```

Output

```
Enter name: Dennis Ritchie
Your name is Dennis.
```

Even though `Dennis Ritchie` was entered in the above program, only `"Dennis"` was stored in the `name` string. It's because there was a space after `Dennis`.

Also notice that we have used the code `name` instead of `&name` with `scanf()`.

```
scanf("%s", name);
```

This is because `name` is a `char` array, and we know that array names decay to pointers in C.

Thus, the `name` in `scanf()` already points to the address of the first element in the string, which is why we don't need to use `&`.

How to read a line of text?

You can use the `fgets()` function to read a line of string. And, you can use `puts()` to display the string.

Example 2: fgets() and puts()

```
#include <stdio.h>
int main()
{
    char name[30];
    printf("Enter name: ");
    fgets(name, sizeof(name), stdin);  // read string
    printf("Name: ");
    puts(name);    // display string
    return 0;
}
```

Output

```
Enter name: Tom Hanks
Name: Tom Hanks
```

Here, we have used `fgets()` function to read a string from the user.

```
fgets(name, sizeof(name), stdlin); // read string
```

The `sizeof(name)` results to 30. Hence, we can take a maximum of 30 characters as input which is the size of the `name` string.

To print the string, we have used `puts(name);`.

Note: The `gets()` function can also be to take input from the user. However, it is removed from the C standard.

It's because `gets()` allows you to input any length of characters. Hence, there might be a buffer overflow.

Passing Strings to Functions

Strings can be passed to a function in a similar way as arrays. Learn more about passing arrays to a function.

Example 3: Passing string to a Function

```
#include <stdio.h>
void displayString(char str[]);

int main()
{
    char str[50];
    printf("Enter string: ");
    fgets(str, sizeof(str), stdin);
    displayString(str);     // Passing string to a function.
    return 0;
}
void displayString(char str[])
{
    printf("String Output: ");
    puts(str);
}
```

Strings and Pointers

Similar like arrays, string names are "decayed" to pointers. Hence, you can use pointers to manipulate elements of the string. We recommended you to check C Arrays and Pointers before you check this example.

Example 4: Strings and Pointers

```c
#include <stdio.h>

int main(void) {
  char name[] = "Harry Potter";

  printf("%c", *name);       // Output: H
  printf("%c", *(name+1));   // Output: a
  printf("%c", *(name+7));   // Output: o

  char *namePtr;

  namePtr = name;
  printf("%c", *namePtr);      // Output: H
  printf("%c", *(namePtr+1));   // Output: a
  printf("%c", *(namePtr+7));   // Output: o
}
```

Commonly Used String Functions

Function	Work of Function
strlen()	computes string's length
strcpy()	copies a string to another
strcat()	concatenates(joins) two strings
strcmp()	compares two strings
strlwr()	converts string to lowercase
strupr()	converts string to uppercase

String Manipulations In C Programming Using Library Functions

To solve this, C supports a large number of string handling functions in the standard library `"string.h"`.

Strings handling functions are defined under `"string.h"` header file.

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

Note: You have to include the code below to run string handling functions.

gets() and puts()

Functions gets() and puts() are two string functions to take string input from the user and display it respectively as mentioned in the previous chapter.

```
#include<stdio.h>

int main()
{
    char name[30];
    printf("Enter name: ");
    gets(name);      //Function to read string from user.
    printf("Name: ");
    puts(name);     //Function to display string.
    return 0;
}
```

Note: Though, `gets()` and `puts()` function handle strings, both these functions are defined in `"stdio.h"` header file.

String Examples

Find the frequency of a character in a string

Find the number of vowels, consonants, digits and white spaces

---

Reverse a string using recursion

Find the length of a string

Concatenate two strings

C Program to Copy a String

Remove all characters in a string except alphabets

C struct

In this tutorial, you'll learn about struct types in C Programming 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.

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;
  ...
};
```

For 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 {
  // code
};

int main() {
  struct Person person1, person2, p[20];
  return 0;
}
```

Another way of creating a `struct` variable is:

```
struct Person {
   // code
} person1, person2, p[20];
```

In both cases,

- `person1` and `person2` are `struct Person` variables
- `p[]` is a `struct Person` array of size 20.

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 1: C++ structs

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

// create struct with person1 variable
struct Person {
  char name[50];
  int citNo;
  float salary;
} person1;

int main() {

  // assign value to name of person1
  strcpy(person1.name, "George Orwell");

  // assign values to other person1 variables
  person1.citNo = 1984;
  person1. salary = 2500;

  // print struct variables
  printf("Name: %s\n", person1.name);
  printf("Citizenship No.: %d\n", person1.citNo);
  printf("Salary: %.2f", person1.salary);

  return 0;
}
```

Output

```
Name: George Orwell
Citizenship No.: 1984
Salary: 2500.00
```

In this program, we have created a `struct` named `Person`. We have also created a variable of `Person` named `person1`.

In `main()`, we have assigned values to the variables defined in `Person` for the `person1` object.

```
strcpy(person1.name, "George Orwell");
person1.citNo = 1984;
person1. salary = 2500;
```

Notice that we have used `strcpy()` function to assign the value to `person1.name`.

This is because `name` is a `char` array (C-string) and we cannot use the assignment operator `=` with it after we have declared the string.

Finally, we printed the data of `person1`.

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.

For example, let us look at the following code:

```
struct Distance{
  int feet;
  float inch;
};

int main() {
  struct Distance d1, d2;
}
```

We can use `typedef` to write an equivalent code with a simplified syntax:

```
typedef struct Distance {
  int feet;
  float inch;
} distances;

int main() {
  distances d1, d2;
}
```

Example 2: C++ typedef

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

// struct with typedef person
typedef struct Person {
  char name[50];
  int citNo;
  float salary;
} person;

int main() {

  // create  Person variable
  person p1;

  // assign value to name of p1
  strcpy(p1.name, "George Orwell");

  // assign values to other p1 variables
  p1.citNo = 1984;
  p1. salary = 2500;

  // print struct variables
  printf("Name: %s\n", p1.name);
  printf("Citizenship No.: %d\n", p1.citNo);
  printf("Salary: %.2f", p1.salary);

  return 0;
}
```

Output

```
Name: George Orwell
Citizenship No.: 1984
Salary: 2500.00
```

Here, we have used `typedef` with the `Person` structure to create an alias `person`.

```
// struct with typedef person
typedef struct Person {
  // code
} person;
```

Now, we can simply declare a `Person` variable using the `person` alias:

```
// equivalent to struct Person p1
person p1;
```

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;
```

Example 3: C++ Nested Structures

```
#include <stdio.h>

struct complex {
  int imag;
  float real;
};

struct number {
  struct complex comp;
  int integer;
} num1;

int main() {

  // initialize complex variables
  num1.comp.imag = 11;
  num1.comp.real = 5.25;

  // initialize number variable
  num1.integer = 6;
      
  // print struct variables
  printf("Imaginary Part: %d\n", num1.comp.imag);
  printf("Real Part: %.2f\n", num1.comp.real);
  printf("Integer: %d", num1.integer);

  return 0;
}
```

Output

```
Imaginary Part: 11
Real Part: 5.25
Integer: 6
```

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 Pointers to struct

Here's how you can create pointers to structs.

```
struct name {
    member1;
    member2;
    .
    .
};

int main()
{
    struct name *ptr, Harry;
}
```

Here, `ptr` is a pointer to `struct`.

Example: Access members using Pointer

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

```
#include <stdio.h>
struct person
{
   int age;
   float weight;
};
```

```c
int main()
{
    struct person *personPtr, person1;
    personPtr = &person1;

    printf("Enter age: ");
    scanf("%d", &personPtr->age);

    printf("Enter weight: ");
    scanf("%f", &personPtr->weight);

    printf("Displaying:\n");
    printf("Age: %d\n", personPtr->age);
    printf("weight: %f", personPtr->weight);

    return 0;
}
```

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.

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

```
#include <stdio.h>
#include <stdlib.h>
struct person {
   int age;
   float weight;
   char name[30];
};

int main()
{
   struct person *ptr;
   int i, n;

   printf("Enter the number of persons: ");
   scanf("%d", &n);

   // allocating memory for n numbers of struct person
   ptr = (struct person*) malloc(n * sizeof(struct person));

   for(i = 0; i < n; ++i)
   {
       printf("Enter first name and age respectively: ");

       // To access members of 1st struct person,
       // ptr->name and ptr->age is used

       // To access members of 2nd struct person,
       // (ptr+1)->name and (ptr+1)->age is used
       scanf("%s %d", (ptr+i)->name, &(ptr+i)->age);
   }

   printf("Displaying Information:\n");
   for(i = 0; i < n; ++i)
       printf("Name: %s\tAge: %d\n", (ptr+i)->name, (ptr+i)->age);

   return 0;
}
```

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,

```
ptr = (struct person*) malloc(n * sizeof(struct person));
```

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

Passing structs to functions

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

- C structures
- C functions
- User-defined Function

Here's how you can pass structures to a function

```
#include <stdio.h>
struct student {
   char name[50];
   int age;
};
```

```
// function prototype
void display(struct student s);

int main() {
    struct student s1;

    printf("Enter name: ");

    // read string input from the user until \n is entered
    // \n is discarded
    scanf("%[^\n]%*c", s1.name);

    printf("Enter age: ");
    scanf("%d", &s1.age);

    display(s1); // passing struct as an argument

    return 0;
}

void display(struct student s) {
    printf("\nDisplaying information\n");
    printf("Name: %s", s.name);
    printf("\nAge: %d", s.age);
}
```

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:

```
#include <stdio.h>
struct student
{
    char name[50];
    int age;
};

// function prototype
struct student getInformation();

int main()
{
    struct student s;

    s = getInformation();

    printf("\nDisplaying information\n");
    printf("Name: %s", s.name);
    printf("\nRoll: %d", s.age);

    return 0;
}
struct student getInformation()
{
  struct student s1;

  printf("Enter name: ");
  scanf ("%[^\n]%*c", s1.name);

  printf("Enter age: ");
  scanf("%d", &s1.age);

  return s1;
}
```

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`.

C struct Examples

Store information of a student using structure

Add two distances (in inch-feet)

Add two complex numbers by passing structures to a function

Store information of n students using structures

Store information of 10 students using structures
