

10History of C Language



History of C language is interesting to know. Here we are going to discuss a brief history of the c language.

C programming language was developed in 1972 by Dennis Ritchie at bell laboratories of AT&T (American Telephone & Telegraph), located in the U.S.A.

Dennis Ritchie is known as the **founder of the c language**.

It was developed to overcome the problems of previous languages such as B, BCPL, etc.

Initially, C language was developed to be used in **UNIX operating system**. It inherits many features of previous languages such as B and BCPL.

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

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

It can be defined by the following ways:

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

1) C as a mother language

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

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

2) C as a system programming language

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

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

3) C as a procedural language

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

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

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

4) C as a structured programming language

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

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

5) C as a mid-level programming language

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

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

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

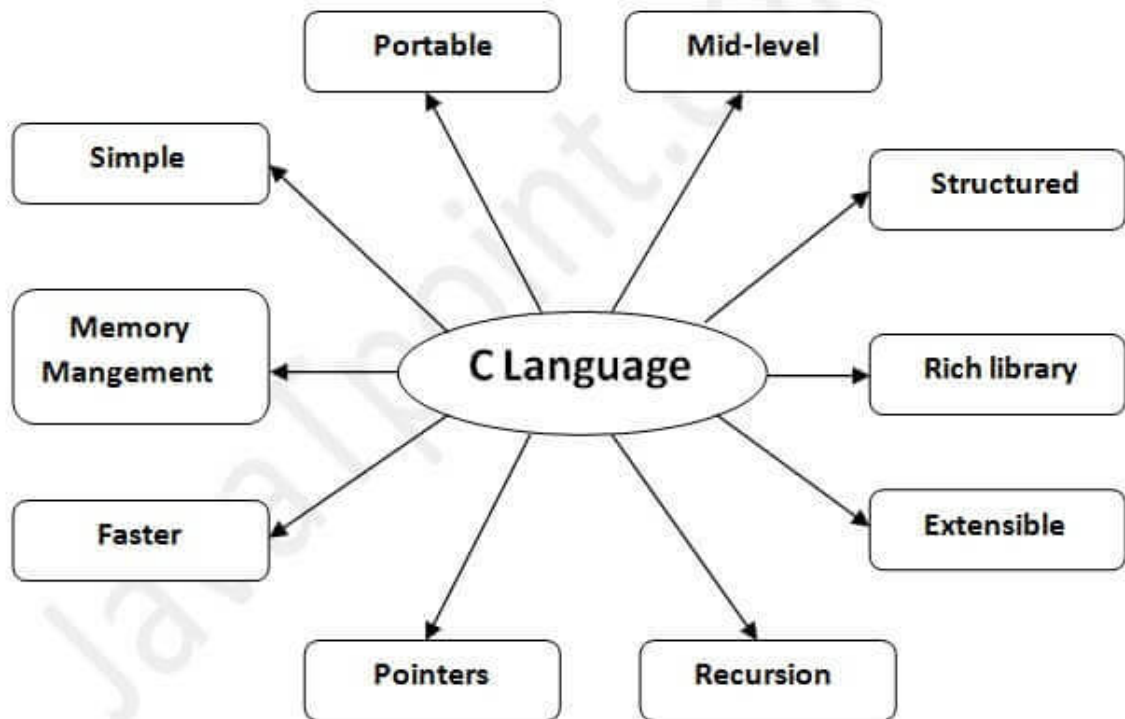
C Program

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

File: main.c

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

Features of C Language



JavaTpoint.com

How to install C

<https://static.javatpoint.com/cpages/software/tc3.zip>

First C Program

Before starting the abcd of C language, you need to learn how to write, compile and run the first c program.

To write the first c program, open the C console and write the following code:

1. `#include <stdio.h>`
2. `int main(){`
3. `printf("Hello C Language");`

4. **return** 0;
5. }

#include <stdio.h> includes the **standard input output** library functions. The **printf()** function is defined in **stdio.h** .

int main() The **main()** function is the entry point of every program in c language.

printf() The **printf()** function is **used to print data** on the console.

return 0 The **return 0** statement, returns execution status to the OS. The 0 value is used for successful execution and 1 for unsuccessful execution.

How to compile and run the c program

There are 2 ways to compile and run the c program, by menu and by shortcut.

By menu

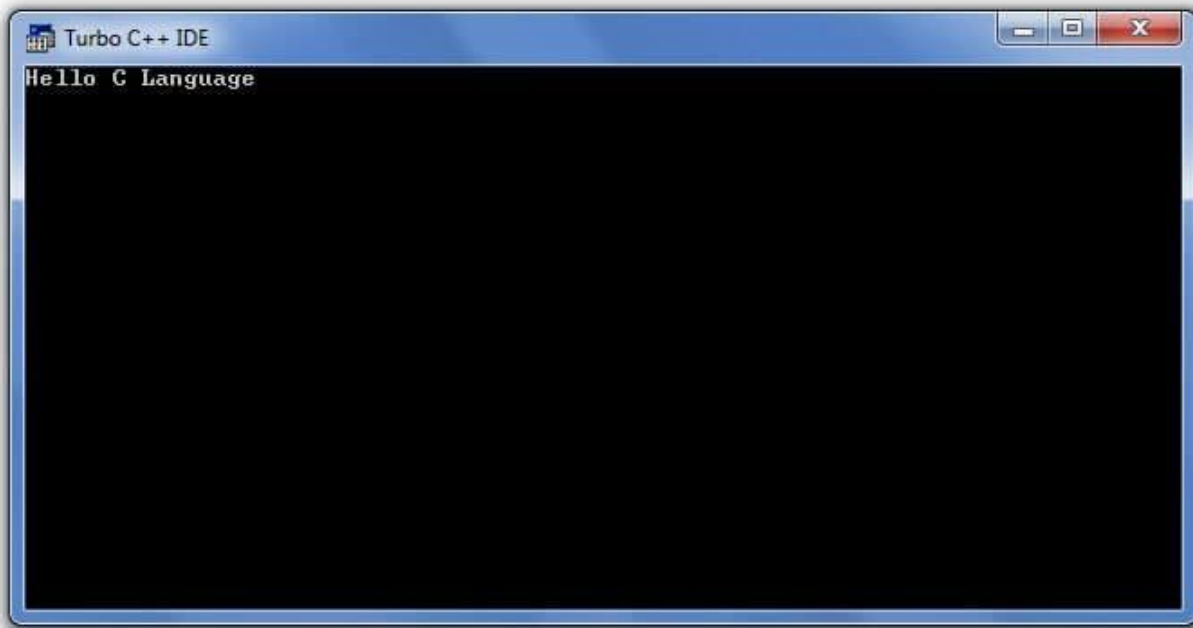
Now **click on the compile menu then compile sub menu** to compile the c program.

Then **click on the run menu then run sub menu** to run the c program.

By shortcut

Or, press ctrl+f9 keys compile and run the program directly.

You will see the following output on user screen.



You can view the user screen any time by pressing the **alt+f5** keys.

Now **press Esc** to return to the turbo c++ console.

Variables in C

A **variable** is a name of the memory location. It is used to store data. Its value can be changed, and it can be reused many times.

It is a way to represent memory location through symbol so that it can be easily identified.

Let's see the syntax to declare a variable:

1. `type variable_list;`

The example of declaring the variable is given below:

1. `int a;`
2. `float b;`
3. `char c;`

Here, a, b, c are variables. The int, float, char are the data types.

We can also provide values while declaring the variables as given below:

1. `int a=10,b=20; //declaring 2 variable of integer type`
2. `float f=20.8;`
3. `char c='A';`

Rules for defining variables

- A variable can have alphabets, digits, and underscore.
- A variable name can start with the alphabet, and underscore only. It can't start with a digit.
- No whitespace is allowed within the variable name.
- A variable name must not be any reserved word or keyword, e.g. int, float, etc.

Valid variable names:

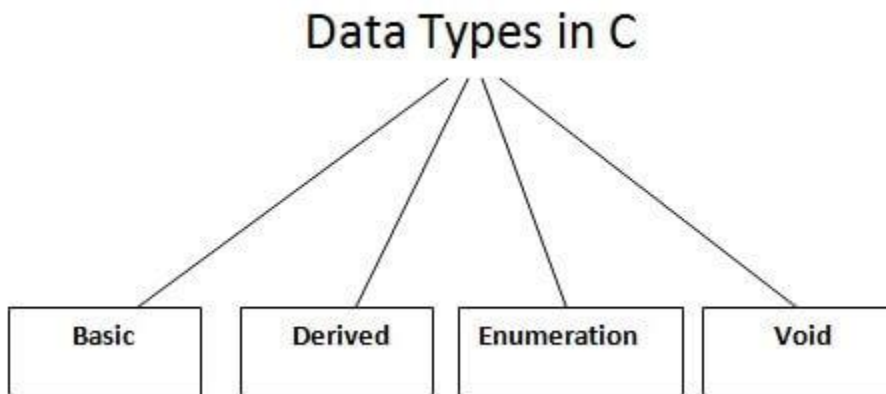
1. `int a;`
2. `int _ab;`
3. `int a30;`

Invalid variable names:

1. `int 2;`
2. `int a b;`
3. `int long;`

Data Types in C

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



There are the following data types in C language.

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

Basic Data Types

The basic data types are integer-based and floating-point based. C language supports both signed and unsigned literals.

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

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

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

float	4 byte	
double	8 byte	
long double	10 byte	

Keywords in C

A keyword is a **reserved word**. You cannot use it as a variable name, constant name, etc. There are only 32 reserved words (keywords) in the C language.

A list of 32 keywords in the c language is given below:

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

C Identifiers

C identifiers represent the name in the C program, for example, variables, functions, arrays, structures, unions, labels, etc. An identifier can be composed of letters such as uppercase, lowercase letters, underscore, digits, but the starting letter should be either an alphabet or an underscore. If the identifier is not used in the external linkage, then it is called as an internal identifier. If the identifier is used in the external linkage, then it is called as an external identifier.

We can say that an identifier is a collection of alphanumeric characters that begins either with an alphabetical character or an underscore, which are used to represent various programming elements such as variables, functions, arrays, structures, unions,

labels, etc. There are 52 alphabetical characters (uppercase and lowercase), underscore character, and ten numerical digits (0-9) that represent the identifiers. There is a total of 63 alphanumerical characters that represent the identifiers.

Rules for constructing C identifiers

- The first character of an identifier should be either an alphabet or an underscore, and then it can be followed by any of the character, digit, or underscore.
- It should not begin with any numerical digit.
- In identifiers, both uppercase and lowercase letters are distinct. Therefore, we can say that identifiers are case sensitive.
- Commas or blank spaces cannot be specified within an identifier.
- Keywords cannot be represented as an identifier.
- The length of the identifiers should not be more than 31 characters.
- Identifiers should be written in such a way that it is meaningful, short, and easy to read.

Example of valid identifiers

1. total, sum, average, _m_, sum_1, etc.

Example of invalid identifiers

1. 2sum (starts with a numerical digit)
2. **int** (reserved word)
3. **char** (reserved word)
4. m+n (special character, i.e., '+')

Types of identifiers

- Internal identifier

- External identifier

Internal Identifier

If the identifier is not used in the external linkage, then it is known as an internal identifier. The internal identifiers can be local variables.

External Identifier

If the identifier is used in the external linkage, then it is known as an external identifier. The external identifiers can be function names, global variables.

Differences between Keyword and Identifier

Keyword	Identifier
Keyword is a pre-defined word.	The identifier is a user-defined word
It must be written in a lowercase letter.	It can be written in both lowercase and uppercase letters.
Its meaning is pre-defined in the c compiler.	Its meaning is not defined in the c compiler.
It is a combination of alphabetical characters.	It is a combination of alphanumeric characters.
It does not contain the underscore character.	It can contain the underscore character.

Let's understand through an example.

1. `int` main()
2. {
3. `int` a=10;

```
4.  int A=20;
5.  printf("Value of a is : %d",a);
6.  printf("\nValue of A is :%d",A);
7.  return 0;
8. }
```

Output

```
Value of a is : 10
Value of A is :20
```

The above output shows that the values of both the variables, 'a' and 'A' are different. Therefore, we conclude that the identifiers are case sensitive.

C Operators

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

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

- Arithmetic Operators
- Relational Operators
- Shift Operators
- Logical Operators
- Bitwise Operators
- Ternary or Conditional Operators
- Assignment Operator

Precedence of Operators in C

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

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

1. `int value=10+20*10;`

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

The precedence and associativity of C operators is given below:

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

Logical AND	&&	Left to right
Logical OR		Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %= >>= <<= &= ^= =	Right to left
Comma	,	Left to right

An operator is a symbol that tells the compiler to perform specific mathematical or logical functions. C language is rich in built-in operators and provides the following types of operators –

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Assignment Operators
- Misc Operators

We will, in this chapter, look into the way each operator works.

Arithmetic Operators

The following table shows all the arithmetic operators supported by the C language. Assume variable A holds 10 and variable B holds 20 then –

Show Examples

Operator	Description	Example
+	Adds two operands.	A + B = 30
-	Subtracts second operand from the first.	A - B = -10

*	Multiplies both operands.	$A * B = 200$
/	Divides numerator by de-numerator.	$B / A = 2$
%	Modulus Operator and remainder of after an integer division.	$B \% A = 0$
++	Increment operator increases the integer value by one.	$A++ = 11$
--	Decrement operator decreases the integer value by one.	$A-- = 9$

Relational Operators

The following table shows all the relational operators supported by C. Assume variable A holds 10 and variable B holds 20 then –

Show Examples

Operator	Description	Example
==	Checks if the values of two operands are equal or not. If yes, then the condition becomes true.	$(A == B)$ is not true.
!=	Checks if the values of two operands are equal or not. If the values are not equal, then the condition becomes true.	$(A != B)$ is true.

>	Checks if the value of left operand is greater than the value of right operand. If yes, then the condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand. If yes, then the condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand. If yes, then the condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand. If yes, then the condition becomes true.	(A <= B) is true.

Logical Operators

Following table shows all the logical operators supported by C language. Assume variable A holds 1 and variable B holds 0, then –

Show Examples

Operator	Description	Example
&&	Called Logical AND operator. If both the operands are non-zero, then the condition becomes true.	(A && B) is false.

	Called Logical OR Operator. If any of the two operands is non-zero, then the condition becomes true.	(A B) is true.
!	Called Logical NOT Operator. It is used to reverse the logical state of its operand. If a condition is true, then Logical NOT operator will make it false.	!(A && B) is true.

Bitwise Operators

Bitwise operator works on bits and perform bit-by-bit operation. The truth tables for &, |, and ^ is as follows –

p	q	p & q	p q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

Assume A = 60 and B = 13 in binary format, they will be as follows –

A = 0011 1100

B = 0000 1101

A&B = 0000 1100

$A|B = 0011\ 1101$

$A^B = 0011\ 0001$

$\sim A = 1100\ 0011$

int main()

{

int a=5,b=4;

/* 8 4 2 1

0 1 0 1=5

0 1 0 0=4

0 1 0 0=4

0 1 0 1=5

-(n+1)

-(4+1)

0 1 0 1=5

0 1 0 1 0 0=3---LEFT SHIFT

0 0 0 1=2---RIGHT SHIFT

*/

printf("result of Bitwise AND=%d\n",a&b);

printf("result of Bitwise OR=%d\n",a|b);

printf("result of Bitwise XOR=%d\n",a^b);

printf("result of Bitwise COMPLEMENT=%d\n",~b);//analys

printf("result of Bitwise LEFT SHIFT=%d\n",a<<2);

printf("result of Bitwise RIGHT SHIFT=%d\n",a>>2);

```
    return 0;
}
```

The following table lists the bitwise operators supported by C. Assume variable 'A' holds 60 and variable 'B' holds 13, then –

Show Examples

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) = 12, i.e., 0000 1100
	Binary OR Operator copies a bit if it exists in either operand.	(A B) = 61, i.e., 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) = 49, i.e., 0011 0001
~	Binary One's Complement Operator is unary and has the effect of 'flipping' bits.	(~A) = ~(60), i.e., -0111101
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 = 240 i.e., 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 = 15 i.e., 0000 1111

Assignment Operators

The following table lists the assignment operators supported by the C language –

Show Examples

Operator	Description	Example
=	Simple assignment operator. Assigns values from right side operands to left side operand	$C = A + B$ will assign the value of $A + B$ to C
+=	Add AND assignment operator. It adds the right operand to the left operand and assign the result to the left operand.	$C += A$ is equivalent to $C = C + A$
-=	Subtract AND assignment operator. It subtracts the right operand from the left operand and assigns the result to the left operand.	$C -= A$ is equivalent to $C = C - A$
*=	Multiply AND assignment operator. It multiplies the right operand with the left operand and assigns the result to the left operand.	$C *= A$ is equivalent to $C = C * A$
/=	Divide AND assignment operator. It divides the left operand with the right operand and assigns the result to the left operand.	$C /= A$ is equivalent to $C = C / A$
%=	Modulus AND assignment operator. It takes modulus using two operands and assigns the result to the left operand.	$C \% = A$ is equivalent to $C = C \% A$

<<=	Left shift AND assignment operator.	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator.	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator.	C &= 2 is same as C = C & 2
^=	Bitwise exclusive OR and assignment operator.	C ^= 2 is same as C = C ^ 2
=	Bitwise inclusive OR and assignment operator.	C = 2 is same as C = C 2

C Format Specifier

The Format specifier is a string used in the formatted input and output functions. The format string determines the format of the input and output. The format string always starts with a '%' character.

The commonly used format specifiers in printf() function are:

Format specifier	Description
%d or %i	It is used to print the signed integer value where signed integer means that the variable can hold both positive and negative values.

%u	It is used to print the unsigned integer value where the unsigned integer means that the variable can hold only positive value.
%o	It is used to print the octal unsigned integer where octal integer value always starts with a 0 value.
%x	It is used to print the hexadecimal unsigned integer where the hexadecimal integer value always starts with a 0x value. In this, alphabetical characters are printed in small letters such as a, b, c, etc.
%X	It is used to print the hexadecimal unsigned integer, but %X prints the alphabetical characters in uppercase such as A, B, C, etc.
%f	It is used for printing the decimal floating-point values. By default, it prints the 6 values after '.'.
%e/%E	It is used for scientific notation. It is also known as Mantissa or Exponent.
%g	It is used to print the decimal floating-point values, and it uses the fixed precision, i.e., the value after the decimal in input would be exactly the same as the value in the output.
%p	It is used to print the address in a hexadecimal form.
%c	It is used to print the unsigned character.
%s	It is used to print the strings.
%ld	It is used to print the long-signed integer value.

Useful Escape Sequences

Escape sequence	Description	Example	Output
<code>\n</code>	New line	<code>printf("Hello \n World");</code>	Hello World
<code>\t</code>	Horizontal tab	<code>printf("Hello \t World");</code>	Hello World
<code>\'</code>	Single quote	<code>printf("Hello \'World\' ");</code>	Hello 'World'
<code>\"</code>	Double quote	<code>printf("Hello \"World\" ");</code>	Hello "World"
<code>\\</code>	Backslash	<code>printf("Hello \\World");</code>	Hello \World

1. `#include<stdio.h>`
2. `int main(){`
3. `int number=50;`
4. `printf("You\nare\nlearning\n\'c\' language\n\"Do you know C language\");`
5. `return 0;`
6. `}`

ASCII value in C

What is ASCII code?

The full form of ASCII is the **American Standard Code for information interchange**. It is a character encoding scheme used for electronics communication. Each character or a special character is represented by some ASCII code, and each ascii code occupies 7 bits in memory.

In **C programming language**, a character variable does not contain a character value itself rather the ascii value of the character variable. The ascii value represents the character variable in numbers, and each character variable is assigned with some number range from 0 to 127. For example, the ascii value of 'A' is 65.

```
1. #include <stdio.h>
2. int main()
3. {
4.     char ch; // variable declaration
5.     printf("Enter a character");
6.     scanf("%c",&ch); // user input
7.     printf("\n The ascii value of the ch variable is : %d", ch);
8.     return 0;
9. }
```

```
1. #include <stdio.h>
2. int main()
3. {
4.     int k; // variable declaration
5.     for(int k=0;k<=255;k++) // for loop from 0-255
6.     {
7.         printf("\nThe ascii value of %c is %d", k,k);
8.     }
```

```
9. return 0;  
10.}
```

Constants in C

A constant is a value or variable that can't be changed in the program, for example: 10, 20, 'a', 3.4, "c programming" etc.

There are different types of constants in C programming.

List of Constants in C

Constant	Example
Decimal Constant	10, 20, 450 etc.
Real or Floating-point Constant	10.3, 20.2, 450.6 etc.
Octal Constant	021, 033, 046 etc.
Hexadecimal Constant	0x2a, 0x7b, 0xaa etc.
Character Constant	'a', 'b', 'x' etc.
String Constant	"c", "c program", "c in javatpoint" etc.

2 ways to define constant in C

There are two ways to define constant in C programming.

1. const keyword
2. #define preprocessor

1) C const keyword

The const keyword is used to define constant in C programming.

1. **const float** PI=3.14;

Now, the value of PI variable can't be changed.

1. **#include**<stdio.h>
2. **int** main(){
3. **const float** PI=3.14;
4. printf("The value of PI is: %f",PI);
5. **return** 0;
6. }

```
#include <stdio.h>
#define PI 3.14
// Driver code
int main()
{
    int n = 4;//single line
    int m=-10;
    int oc=1100;
    printf("octal value:%o\n",oc);
    printf("Bitwise complement of %d : %d\n",n, ~n);
    printf("Bitwise complement of %u\n",m);

    printf("Bi\tw\tise\ncolbmple\tme\n");
    char ch='A';
    printf("%d",ch);

    //const float PI=3.14;
    printf("%f",PI);
    return 0;
}
```

What are literals?

Literals are the constant values assigned to the constant variables. We can say that the literals represent the fixed values that cannot be modified. It also contains memory but does not have references as variables. For example, `const int =10;` is a constant integer expression in which 10 is an integer literal.

Types of literals

There are four types of literals that exist in C programming

:

- **Integer literal**
- **Float literal**
- **Character literal**
- **String literal**

Integer literal

It is a numeric literal that represents only integer type values. It represents the value neither in fractional nor exponential part.

It can be specified in the following three ways:

Decimal number (base 10)

It is defined by representing the digits between 0 to 9. For example, 45, 67, etc.

Octal number (base 8)

It is defined as a number in which 0 is followed by digits such as 0,1,2,3,4,5,6,7. For example, 012, 034, 055, etc.

Hexadecimal number (base 16)

It is defined as a number in which 0x or 0X is followed by the hexadecimal digits (i.e., digits from 0 to 9, alphabetical characters from (a-z) or (A-Z)).

An integer literal is suffixed by following two sign qualifiers:

L or l: It is a size qualifier that specifies the size of the integer type as long.

U or u: It is a sign qualifier that represents the type of the integer as unsigned. An unsigned qualifier contains only positive values.

Note: The order of the qualifier is not considered, i.e., both lu and ul are the same.

Let's look at a simple example of integer literal.

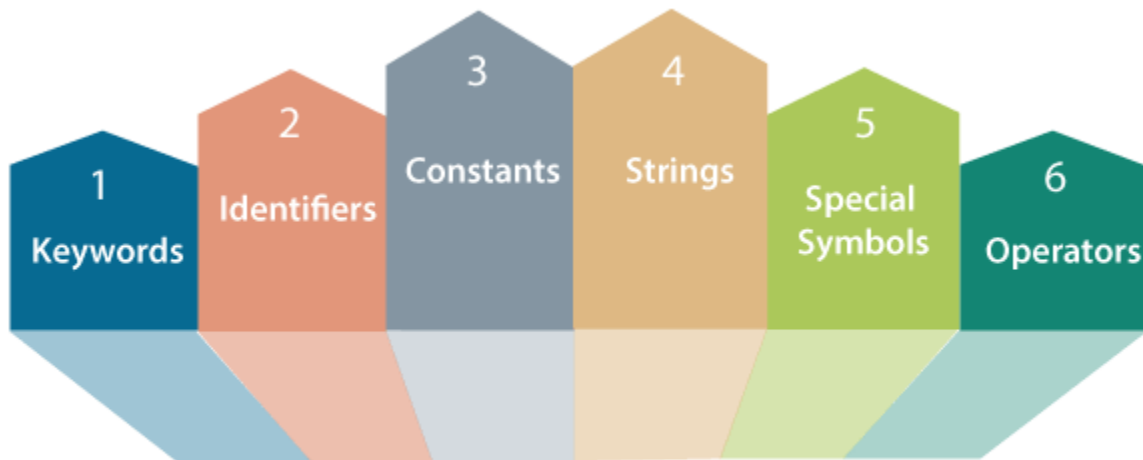
1. `#include <stdio.h>`
2. `int main()`
3. `{`
4. `const int a=23; // constant integer literal`
5. `printf("Integer literal : %d", a);`
6. `return 0;`
7. `}`

Tokens in C

Tokens in C is the most important element to be used in creating a program in C. We can define the token as the smallest individual element in C. For example, we cannot create a sentence without using words; similarly, we cannot create a program in C without using tokens in C. Therefore, we can say that tokens in C is the building block or the basic component for creating a program in C language.

Classification of tokens in C

Tokens in C language can be divided into the following categories:



Classification of C Tokens

- Keywords in C
- Identifiers in C
- Strings in C
- Operators in C
- Constant in C
- Special Characters in C

Let's understand each token one by one.

Keywords in C

Keywords in C can be defined as the **pre-defined** or the **reserved words** having its own importance, and each keyword has its own functionality. Since keywords are the pre-defined words used by the compiler, so they cannot be used as the variable names. If the keywords are used as the variable names, it means that we are assigning a different meaning to the keyword, which is not allowed. C language supports 32 keywords given below:

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

Identifiers in C

Identifiers in C are used for naming variables, functions, arrays, structures, etc. Identifiers in C are the user-defined words. It can be composed of uppercase letters, lowercase letters, underscore, or digits, but the starting letter should be either an underscore or an alphabet. Identifiers cannot be used as keywords. Rules for constructing identifiers in C are given below:

- The first character of an identifier should be either an alphabet or an underscore, and then it can be followed by any of the character, digit, or underscore.
- It should not begin with any numerical digit.
- In identifiers, both uppercase and lowercase letters are distinct. Therefore, we can say that identifiers are case sensitive.
- Commas or blank spaces cannot be specified within an identifier.
- Keywords cannot be represented as an identifier.
- The length of the identifiers should not be more than 31 characters.

- Identifiers should be written in such a way that it is meaningful, short, and easy to read.

Strings in C

Strings in C are always represented as an array of characters having null character '\0' at the end of the string. This null character denotes the end of the string. Strings in C are enclosed within double quotes, while characters are enclosed within single characters. The size of a string is a number of characters that the string contains.

Now, we describe the strings in different ways:

```
char a[10] = "javatpoint"; // The compiler allocates the 10 bytes to the 'a' array.
```

```
char a[] = "javatpoint"; // The compiler allocates the memory at the run time.
```

```
char a[10] = {'j','a','v','a','t','p','o','i','n','t','\0'}; // String is represented in the form of characters.
```

Operators in C

Operators in C is a special symbol used to perform the functions. The data items on which the operators are applied are known as operands. Operators are applied between the operands. Depending on the number of operands, operators are classified as follows:

Unary Operator

A unary operator is an operator applied to the single operand. For example: increment operator (++), decrement operator (--), sizeof, (type)*.

Binary Operator

The binary operator is an operator applied between two operands. The following is the list of the binary operators:

- Arithmetic Operators
- Relational Operators
- Shift Operators
- Logical Operators

- Bitwise Operators
- Conditional Operators
- Assignment Operator

Constants in C

A constant is a value assigned to the variable which will remain the same throughout the program, i.e., the constant value cannot be changed.

There are two ways of declaring constant:

- Using const keyword
- Using #define pre-processor

Types of constants in C

Constant	Example
Integer constant	10, 11, 34, etc.
Floating-point constant	45.6, 67.8, 11.2, etc.
Octal constant	011, 088, 022, etc.
Hexadecimal constant	0x1a, 0x4b, 0x6b, etc.
Character constant	'a', 'b', 'c', etc.
String constant	"java", "c++", ".net", etc.

Special characters in C

Some special characters are used in C, and they have a special meaning which cannot be used for another purpose.

- **Square brackets []:** The opening and closing brackets represent the single and multidimensional subscripts.
- **Simple brackets ():** It is used in function declaration and function calling. For example, printf() is a pre-defined function.
- **Curly braces { }:** It is used in the opening and closing of the code. It is used in the opening and closing of the loops.
- **Comma (,):** It is used for separating for more than one statement and for example, separating function parameters in a function call, separating the variable when printing the value of more than one variable using a single printf statement.
- **Hash/pre-processor (#):** It is used for pre-processor directive. It basically denotes that we are using the header file.
- **Asterisk (*):** This symbol is used to represent pointers and also used as an operator for multiplication.
- **Tilde (~):** It is used as a destructor to free memory.
- **Period (.):** It is used to access a member of a structure or a union.

C Boolean

In C, Boolean is a data type that contains two types of values, i.e., 0 and 1. Basically, the bool type value represents two types of behavior, either true or false. Here, '0' represents false value, while '1' represents true value.

In C Boolean, '0' is stored as 0, and another integer is stored as 1. We do not require to use any header file to use the Boolean data type in C++, but in C, we have to use the header file, i.e., stdbool.h. If we do not use the header file, then the program will not compile.

Syntax

1. **bool** variable_name;

In the above syntax, **bool** is the data type of the variable, and **variable_name** is the name of the variable.

Let's understand through an example.

Competitive questions on Structures in Hindi

```
1. #include <stdio.h>
2. #include<stdbool.h>
3. int main()
4. {
5.     bool x=false; // variable initialization.
6.     if(x==true) // conditional statements
7.     {
8.         printf("The value of x is true");
9.     }
10. else
11. printf("The value of x is FALSE");
12. return 0;
13.}
```

In the above code, we have used **<stdbool.h>** header file so that we can use the bool type variable in our program. After the declaration of the header file, we create the bool type variable 'x' and assigns a 'false' value to it. Then, we add the conditional statements, i.e., **if..else**, to determine whether the value of 'x' is true or not.

Output

The value of x is FALSE

Boolean Array

Now, we create a bool type array. The Boolean array can contain either true or false value, and the values of the array can be accessed with the help of indexing.

Let's understand this scenario through an example.

```
1. #include <stdio.h>
2. #include<stdbool.h>
3. int main()
4. {
5.     bool b[2]={true,false}; // Boolean type array
6.     for(int i=0;i<2;i++) // for loop
7.     {
8.         printf("%d",b[i]); // printf statement
9.     }
10. return 0;
11.}
```

In the above code, we have declared a Boolean type array containing two values, i.e., true and false.

Output

1,0



typedef

There is another way of using Boolean value, i.e., **typedef**. Basically, typedef is a keyword in C language, which is used to assign the name to the already existing datatype.

Let's see a simple example of typedef.

```
1. #include <stdio.h>
2. typedef enum{false,true} b;
```

```
3. int main()
4. {
5.  b x=false; // variable initialization
6.  if(x==true) // conditional statements
7.  {
8.  printf("The value of x is true");
9.  }
10. else
11. {
12. printf("The value of x is false");
13. }
14. return 0;
15. }
```

In the above code, we use the Boolean values, i.e., true and false, but we have not used the bool type. We use the Boolean values by creating a new name of the 'bool' type. In order to achieve this, **the typedef** keyword is used in the program.

```
1. typedef enum{false,true} b;
```

The above statement creates a new name for the '**bool**' type, i.e., 'b' as 'b' can contain either true or false value. We use the 'b' type in our program and create the 'x' variable of type 'b'.

Output

The value of x is false

Boolean with Logical Operators

The Boolean type value is associated with logical operators. There are three types of logical operators in the C language:

&&(AND Operator): It is a logical operator that takes two operands. If the value of both the operands are true, then this operator returns true otherwise false

||(OR Operator): It is a logical operator that takes two operands. If the value of both the operands is false, then it returns false otherwise true.

!(NOT Operator): It is a NOT operator that takes one operand. If the value of the operand is false, then it returns true, and if the value of the operand is true, then it returns false.

Let's understand through an example.

```
1. #include <stdio.h>
2. #include<stdbool.h>
3. int main()
4. {
5.     bool x=false;
6.     bool y=true;
7.     printf("The value of x&&y is %d", x&&y);
8.     printf("\nThe value of x||y is %d", x||y);
9.     printf("\nThe value of !x is %d", !x);
10.}
```

Static in C

Static is a keyword used in C programming language. It can be used with both variables and functions, i.e., we can declare a static variable and static function as well. An ordinary variable is limited to the scope in which it is defined, while the scope of the static variable is throughout the program.

Static keyword can be used in the following situations:

- **Static global variable**

When a global variable is declared with a static keyword, then it is known as a

static global variable. It is declared at the top of the program, and its visibility is throughout the program.

- **Static function**

When a function is declared with a static keyword known as a static function. Its lifetime is throughout the program.

- **Static local variable**

When a local variable is declared with a static keyword, then it is known as a static local variable. The memory of a static local variable is valid throughout the program, but the scope of visibility of a variable is the same as the automatic local variables. However, when the function modifies the static local variable during the first function call, then this modified value will be available for the next function call also.

- **Static member variables**

When the member variables are declared with a static keyword in a class, then it is known as static member variables. They can be accessed by all the instances of a class, not with a specific instance.

- **Static method**

The member function of a class declared with a static keyword is known as a static method. It is accessible by all the instances of a class, not with a specific instance.

Let's understand through an example.

1. `#include <stdio.h>`
2. `int main()`
3. `{`
4. `printf("%d",func());`
5. `printf("\n%d",func());`
6. `return 0;`

```
7. }
8. int func()
9. {
10. int count=0; // variable initialization
11. count++; // incrementing counter variable
12.
13. return count; }
```

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

```
int func();
```

```
static int count=0;
```

```
int main()
```

```
{
```

```
printf("%d",func());
```

```
printf("\n%d",func());
```

```
return 0;
```

```
}
```

```
int func()
```

```
{
```

```
// static int count=0; // variable initialization
```

```
count++; // incrementing counter variable 1
```



```
    return count;
}
```

In the above code, the func() function is called. In func(), count variable gets updated. As soon as the function completes its execution, the memory of the count variable will be removed. If we do not want to remove the count from memory, then we need to use the count variable as static. If we declare the variable as static, then the variable will not be removed from the memory even when the function completes its execution.

Output

```
1
1
```



Static variable

A static variable is a variable that persists its value across the various function calls.

Syntax

The syntax of a static variable is given below:

1. **static** data_type variable_name;

Let's look at a simple example of static variable.

1. **#include <stdio.h>**
2. **int** main()

```

3. {
4.   printf("%d",func());
5.   printf("\n%d",func());
6.
7.   return 0;
8. }
9. int func()
10.{
11.   static int count=0;
12.   count++;
13.   return count;
14.}

```

In the above code, we have declared the count variable as static. When the func() is called, the value of count gets updated to 1, and during the next function call, the value of the count variable becomes 2. Therefore, we can say that the value of the static variable persists within the function call.

Output

```

1
2

```

Static Function

As we know that non-static functions are global by default means that the function can be accessed outside the file also, but if we declare the function as static, then it limits the function scope. The static function can be accessed within a file only.

The static function would look like as:

```

1. static void func()
2. {
3.   printf("Hello javaTpoint");

```

4. }

Differences b/w static and global variable

Global variables are the variables that are declared outside the function. These global variables exist at the beginning of the **program**, and its scope remains till the end of the program. It can be accessed outside the program also.

Static variables are limited to the source file in which they are defined, i.e., they are not accessible by the other source files.

Both the static and global variables have static initialization. Here, static initialization means if we do not assign any value to the variable then by default, 0 value will be assigned to the variable.

Differences b/w static local and static global variable

Static global variable

If the variable declared with a static keyword outside the function, then it is known as a static global variable. It is accessible throughout the program.

Static local variable

The variable with a static keyword is declared inside a function is known as a static local variable. The scope of the static local variable will be the same as the automatic local variables, but its memory will be available throughout the program execution. When the function modifies the value of the static local variable during one function call, then it will remain the same even during the next function call.

Properties of a static variable

The following are the properties of a static variable:

- The memory of a static variable is allocated within a static variable.
- Its memory is available throughout the program, but the scope will remain the same as the automatic local variables. Its
- value will persist across the various function calls.

- If we do not assign any value to the variable, then the default value will be 0.
- A global static variable cannot be accessed outside the program, while a global variable can be accessed by other source files.

Programming Errors in C

Errors are the problems or the faults that occur in the program, which makes the behavior of the program abnormal, and experienced developers can also make these faults. Programming errors are also known as the bugs or faults, and the process of removing these bugs is known as **debugging**.

These errors are detected either during the time of compilation or execution. Thus, the errors must be removed from the program for the successful execution of the program.

There are mainly five types of errors exist in C programming:

- **Syntax error**
- **Run-time error**
- **Linker error**
- **Logical error**
- **Semantic error**

Types of errors



Syntax error

Syntax errors are also known as the compilation errors as they occurred at the compilation time, or we can say that the syntax errors are thrown by the compilers. These errors are mainly occurred due to the mistakes while typing or do not follow the syntax of the specified programming language. These mistakes are generally made by beginners only because they are new to the language. These errors can be easily debugged or corrected.

For example:

1. If we want to declare the variable of type integer,
2. `int a; // this is the correct form`
3. `Int a; // this is an incorrect form.`

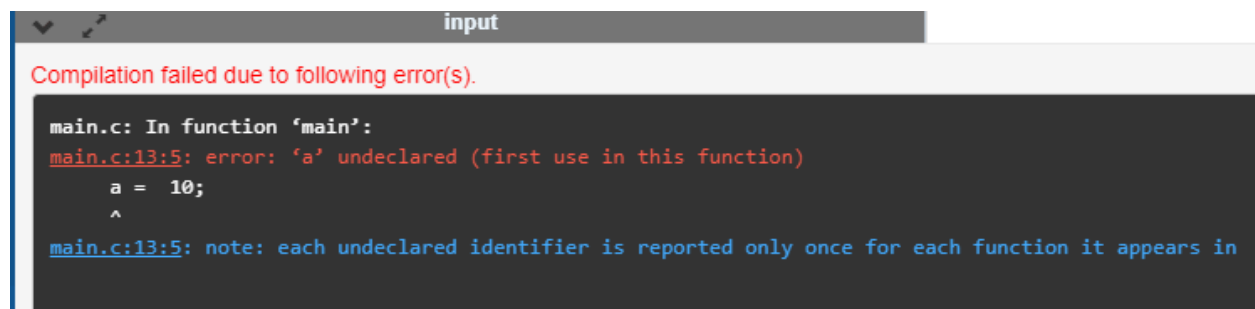
Commonly occurred syntax errors are:

- If we miss the parenthesis (}) while writing the code.
- Displaying the value of a variable without its declaration.
- If we miss the semicolon (;) at the end of the statement.

Let's understand through an example.

1. `#include <stdio.h>`
2. `int main()`
3. `{`
4. `a = 10;`
5. `printf("The value of a is : %d", a);`
6. `return 0;`
7. `}`

Output



```
input
Compilation failed due to following error(s).

main.c: In function 'main':
main.c:13:5: error: 'a' undeclared (first use in this function)
    a = 10;
    ^
main.c:13:5: note: each undeclared identifier is reported only once for each function it appears in
```

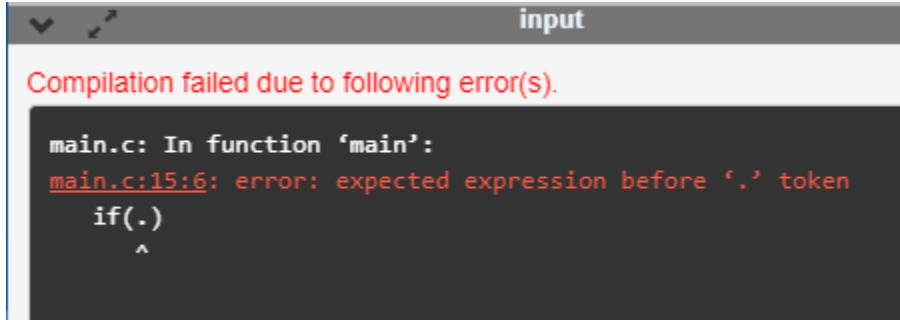
In the above output, we observe that the code throws the error that 'a' is undeclared. This error is nothing but the syntax error only.

There can be another possibility in which the syntax error can exist, i.e., if we make mistakes in the basic construct. Let's understand this scenario through an example.

1. `#include <stdio.h>`
2. `int main()`
3. `{`
4. `int a=2;`
5. `if(.) // syntax error`
6.
7. `printf("a is greater than 1");`
8. `return 0;`
9. `}`

In the above code, we put the (.) instead of condition in 'if', so this generates the syntax error as shown in the below screenshot.

Output



```
input
Compilation failed due to following error(s).
main.c: In function 'main':
main.c:15:6: error: expected expression before '.' token
    if(.)
       ^
```

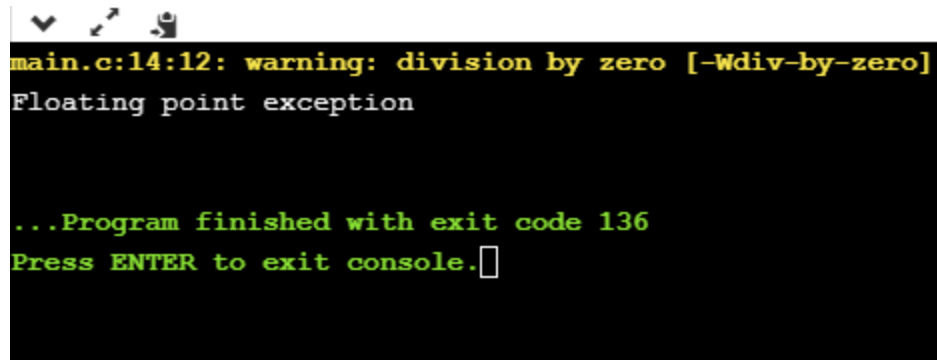
Run-time error

Sometimes the errors exist during the execution-time even after the successful compilation known as run-time errors. When the program is running, and it is not able to perform the operation is the main cause of the run-time error. The division by zero is the common example of the run-time error. These errors are very difficult to find, as the compiler does not point to these errors.

Let's understand through an example.

1. `#include <stdio.h>`
2. `int main()`
3. `{`
4. `int a=2;`
5. `int b=2/0;`
6. `printf("The value of b is : %d", b);`
7. `return 0;`
8. `}`

Output

A screenshot of a terminal window with a black background. At the top, there are three small icons: a downward arrow, a leftward arrow, and a document icon. The terminal text is as follows:

```
main.c:14:12: warning: division by zero [-Wdiv-by-zero]
Floating point exception

...Program finished with exit code 136
Press ENTER to exit console.□
```

In the above output, we observe that the code shows the run-time error, i.e., division by zero.

Linker error

Linker errors are mainly generated when the executable file of the program is not created. This can be happened either due to the wrong function prototyping or usage of the wrong header file. For example, the **main.c** file contains the **sub()** function whose declaration and definition is done in some other file such as **func.c**. During the compilation, the compiler finds the **sub()** function in **func.c** file, so it generates two object files, i.e., **main.o** and **func.o**. At the execution time, if the definition of **sub()** function is not found in the **func.o** file, then the linker error will be thrown. The most common linker error that occurs is that we use **Main()** instead of **main()**.

Let's understand through a simple example.

1. `#include <stdio.h>`
2. `int Main()`
3. `{`
4. `int a=78;`
5. `printf("The value of a is : %d", a);`
6. `return 0;`
7. `}`

Output


```
(.text+0x20): undefined reference to `main'  
collect2: error: ld returned 1 exit status
```

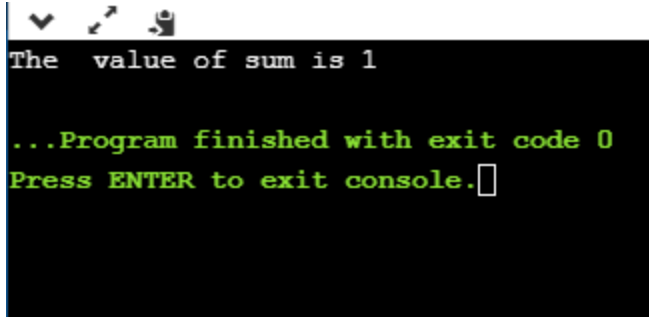
Logical error

The logical error is an error that leads to an undesired output. These errors produce the incorrect output, but they are error-free, known as logical errors. These types of mistakes are mainly done by beginners. The occurrence of these errors mainly depends upon the logical thinking of the developer. If the programmers sound logically good, then there will be fewer chances of these errors.

Let's understand through an example.

```
1. #include <stdio.h>  
2. int main()  
3. {  
4.     int sum=0; // variable initialization  
5.     int k=1;  
6.     for(int i=1;i<=10;i++); // logical error, as we put the semicolon after loop  
7.     {  
8.         sum=sum+k;  
9.         k++;  
10.    }  
11. printf("The value of sum is %d", sum);  
12. return 0;  
13.}
```

Output

A screenshot of a console window with a black background and white and green text. At the top, there are three small icons: a checkmark, a cursor, and a person. The text in the console reads: "The value of sum is 1" in white, followed by "...Program finished with exit code 0" and "Press ENTER to exit console." in green. A white cursor is positioned at the end of the last line.

```
The value of sum is 1
...Program finished with exit code 0
Press ENTER to exit console.
```

In the above code, we are trying to print the sum of 10 digits, but we got the wrong output as we put the semicolon (;) after the for loop, so the inner statements of the for loop will not execute. This produces the wrong output.

Semantic error

Semantic errors are the errors that occurred when the statements are not understandable by the compiler.

The following can be the cases for the semantic error:

- Use of a un-initialized variable.

```
int i;
```

```
i=i+2;
```

- Type compatibility

```
int b = "javatpoint";
```

- Errors in expressions

```
int a, b, c;
```

```
a+b = c;
```

- Array index out of bound

```
int a[10];
```

```
a[10] = 34;
```

Let's understand through an example.

1. `#include <stdio.h>`

```
2. int main()
3. {
4. int a,b,c;
5. a=2;
6. b=3;
7. c=1;
8. a+b=c; // semantic error
9. return 0;
10.}
```

In the above code, we use the statement **a+b =c**, which is incorrect as we cannot use the two operands on the left-side.

Compile time vs Runtime

Compile-time and Runtime are the two programming terms used in the software development. Compile-time is the time at which the source code is converted into an executable code while the run time is the time at which the executable code is started running. Both the compile-time and runtime refer to different types of error.

Compile-time errors

Compile-time errors are the errors that occurred when we write the wrong syntax. If we write the wrong syntax or semantics of any programming language, then the compile-time errors will be thrown by the compiler. The compiler will not allow to run the program until all the errors are removed from the program. When all the errors are removed from the program, then the compiler will generate the executable file.

The compile-time errors can be:

Syntax errors

Semantic errors

Syntax errors

When the programmer does not follow the syntax of any programming language, then the compiler will throw the syntax error.

For example,

```
int a, b:
```

The above declaration generates the compile-time error as in C, every statement ends with the semicolon, but we put a colon (:) at the end of the statement.

Semantic errors

The semantic errors exist when the statements are not meaningful to the compiler.

For example,

```
a+b=c;
```

The above statement throws a compile-time errors. In the above statement, we are assigning the value of 'c' to the summation of 'a' and 'b' which is not possible in C programming language as it can contain only one variable on the left of the assignment operator while right of the assignment operator can contain more than one variable.

The above statement can be re-written as:

```
c=a+b;
```

Runtime errors

The runtime errors are the errors that occur during the execution and after compilation. The examples of runtime errors are division by zero, etc. These errors are not easy to detect as the compiler does not point to these errors.

Let's look at the differences between compile-time and runtime:

Compile-time	Runtime
--------------	---------

The compile-time errors are the errors which are produced at the compile-time, and they are detected by the compiler.

The runtime errors are the errors which are not generated by the compiler and produce an unpredictable result at the execution time.

In this case, the compiler prevents the code from execution if it detects an error in the program.

In this case, the compiler does not detect the error, so it cannot prevent the code from the execution.

It contains the syntax and semantic errors such as missing semicolon at the end of the statement.

It contains the errors such as division by zero, determining the square root of a negative number.

Example of Compile-time error

```
#include <stdio.h>
int main()
{
    int a=20;
    printf("The value of a is : %d",a):
    return 0;
}
```

In the above code, we have tried to print the value of 'a', but it throws an error. We put the colon at the end of the statement instead of a semicolon, so this code generates a compile-time error.

Output

Compile time vs Runtime

Example of runtime error

```
#include <stdio.h>
int main()
{
    int a=20;
    int b=a/0; // division by zero
    printf("The value of b is : %d",b):
    return 0;
}
```

In the above code, we try to divide the value of 'b' by zero, and this throws a runtime error.

Output

Compile time vs Runtime

Conditional Operator in C

The conditional operator is also known as a **ternary operator**. The conditional statements are the decision-making statements which depends upon the output of the expression. It is represented by two symbols, i.e., '?' and ':'.

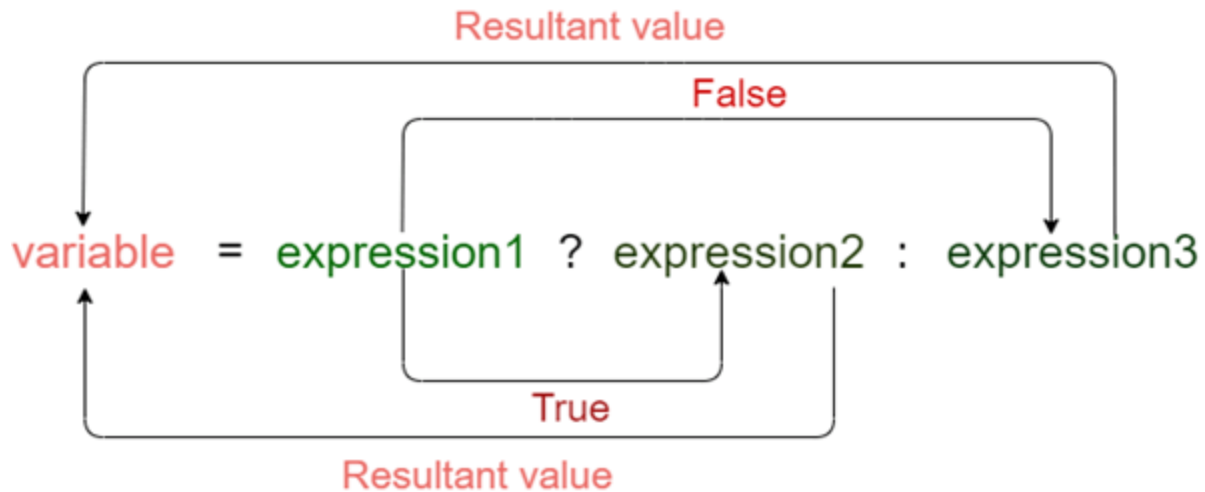
As conditional operator works on three operands, so it is also known as the ternary operator.

The behavior of the conditional operator is similar to the 'if-else' statement as 'if-else' statement is also a decision-making statement.

Syntax of a conditional operator

1. Expression1? expression2: expression3;

The pictorial representation of the above syntax is shown below:



Meaning of the above syntax.

- In the above syntax, the expression1 is a Boolean condition that can be either true or false value.
- If the expression1 results into a true value, then the expression2 will execute.
- The expression2 is said to be true only when it returns a non-zero value.
- If the expression1 returns false value then the expression3 will execute.
- The expression3 is said to be false only when it returns zero value.

Let's understand the ternary or conditional operator through an example.

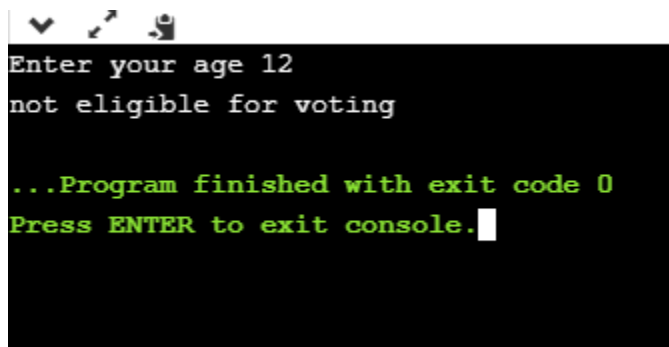
1. `#include <stdio.h>`
2. `int main()`
3. `{`
4. `int age; // variable declaration`
5. `printf("Enter your age");`
6. `scanf("%d",&age); // taking user input for age variable`
7. `(age>=18)? (printf("eligible for voting")) : (printf("not eligible for voting")); // conditional operator`
8. `return 0;`

9. }

In the above code, we are taking input as the 'age' of the user. After taking input, we have applied the condition by using a conditional operator. In this condition, we are checking the age of the user. If the age of the user is greater than or equal to 18, then the statement1 will execute, i.e., (printf("eligible for voting")) otherwise, statement2 will execute, i.e., (printf("not eligible for voting")).

Let's observe the output of the above program.

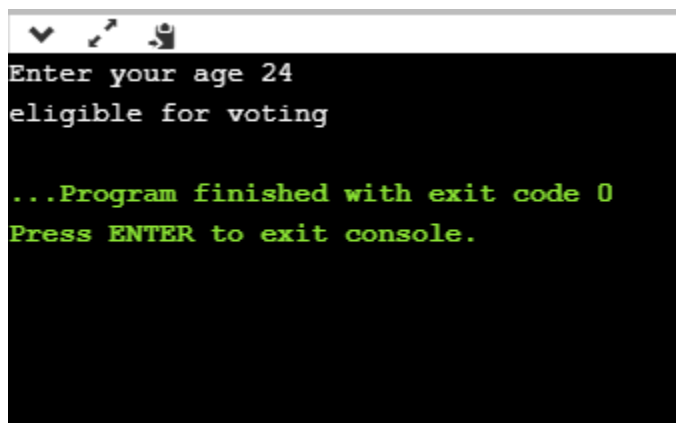
If we provide the age of user below 18, then the output would be:

A screenshot of a terminal window with a black background and white text. At the top, there are three small icons: a downward arrow, a magnifying glass, and a person icon. The text in the terminal reads: "Enter your age 12", "not eligible for voting", "...Program finished with exit code 0", and "Press ENTER to exit console." with a white cursor at the end of the last line.

```
Enter your age 12
not eligible for voting

...Program finished with exit code 0
Press ENTER to exit console.
```

If we provide the age of user above 18, then the output would be:

A screenshot of a terminal window with a black background and white text. At the top, there are three small icons: a downward arrow, a magnifying glass, and a person icon. The text in the terminal reads: "Enter your age 24", "eligible for voting", "...Program finished with exit code 0", and "Press ENTER to exit console." with a white cursor at the end of the last line.

```
Enter your age 24
eligible for voting

...Program finished with exit code 0
Press ENTER to exit console.
```

As we can observe from the above two outputs that if the condition is true, then the statement1 is executed; otherwise, statement2 will be executed.

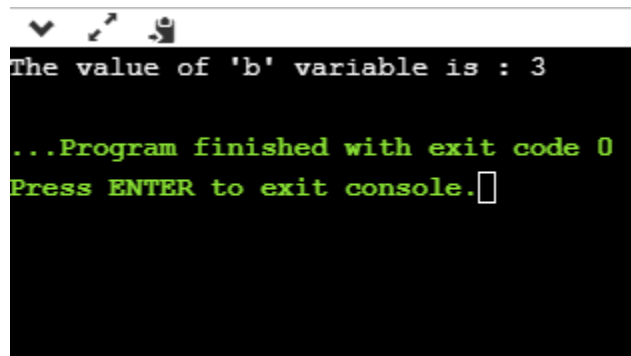
Till now, we have observed that how conditional operator checks the condition and based on condition, it executes the statements. Now, we will see how a conditional operator is used to assign the value to a variable.

Let's understand this scenario through an example.

```
1. #include <stdio.h>
2. int main()
3. {
4.     int a=5,b; // variable declaration
5.     b=((a==5)?(3):(2)); // conditional operator
6.     printf("The value of 'b' variable is : %d",b);
7.     return 0;
8. }
```

In the above code, we have declared two variables, i.e., 'a' and 'b', and assign 5 value to the 'a' variable. After the declaration, we are assigning value to the 'b' variable by using the conditional operator. If the value of 'a' is equal to 5 then 'b' is assigned with a 3 value otherwise 2.

Output



```
The value of 'b' variable is : 3
...Program finished with exit code 0
Press ENTER to exit console.
```

The above output shows that the value of 'b' variable is 3 because the value of 'a' variable is equal to 5.

As we know that the behavior of conditional operator and 'if-else' is similar but they have some differences. Let's look at their differences.

- A conditional operator is a single programming statement, while the 'if-else' statement is a programming block in which statements come under the parenthesis.

- A conditional operator can also be used for assigning a value to the variable, whereas the 'if-else' statement cannot be used for the assignment purpose.
- It is not useful for executing the statements when the statements are multiple, whereas the 'if-else' statement proves more suitable when executing multiple statements.
- The nested ternary operator is more complex and cannot be easily debugged, while the nested 'if-else' statement is easy to read and maintain.

C if else Statement

The if-else statement in C is used to perform the operations based on some specific condition. The operations specified in if block are executed if and only if the given condition is true.

There are the following variants of if statement in C language.

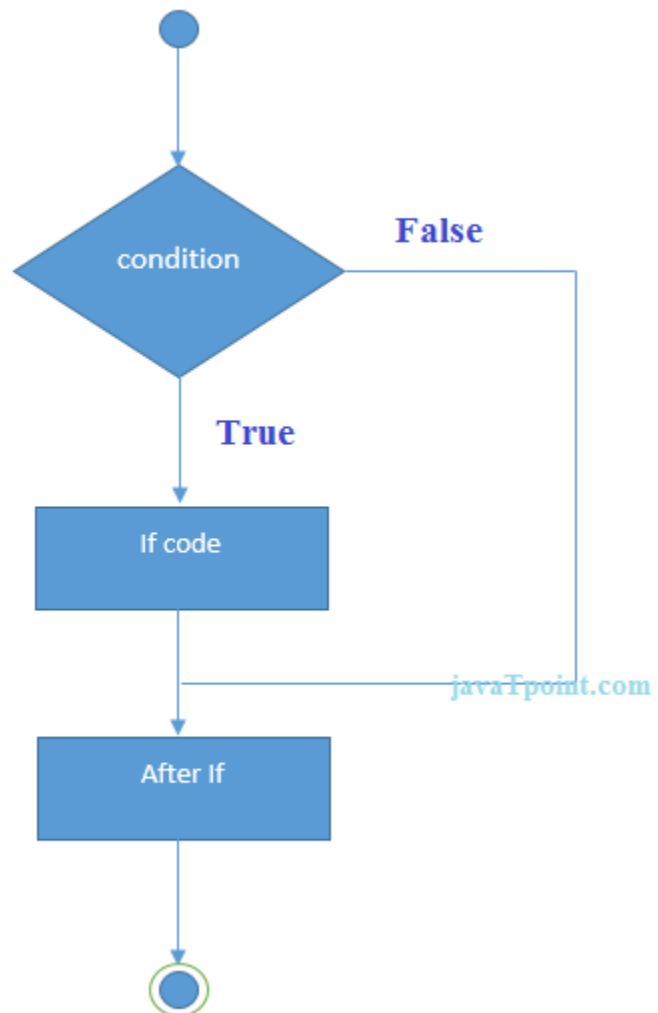
- If statement
- If-else statement
- If else-if ladder
- Nested if

If Statement

The if statement is used to check some given condition and perform some operations depending upon the correctness of that condition. It is mostly used in the scenario where we need to perform the different operations for the different conditions. The syntax of the if statement is given below.

1. `if(expression){`
2. `//code to be executed`
3. `}`

Flowchart of if statement in C



Let's see a simple example of C language if statement.

```
1. #include<stdio.h>
2. int main(){
3.     int number=0;
4.     printf("Enter a number:");
5.     scanf("%d",&number);
6.     if(number%2==0){
7.         printf("%d is even number",number);
8.     }
```

```
9. return 0;
10.}
```

Output

```
Enter a number:4
4 is even number
enter a number:5
```

Program to find the largest number of the three.

```
1. #include <stdio.h>
2. int main()
3. {
4.     int a, b, c;
5.     printf("Enter three numbers?");
6.     scanf("%d %d %d",&a,&b,&c);
7.     if(a>b && a>c)
8.     {
9.         printf("%d is largest",a);
10.    }
11.    if(b>a && b > c)
12.    {
13.        printf("%d is largest",b);
14.    }
15.    if(c>a && c>b)
16.    {
17.        printf("%d is largest",c);
18.    }
19.    if(a == b && a == c)
20.    {
```

```
21.     printf("All are equal");
22. }
23. }
```

Output

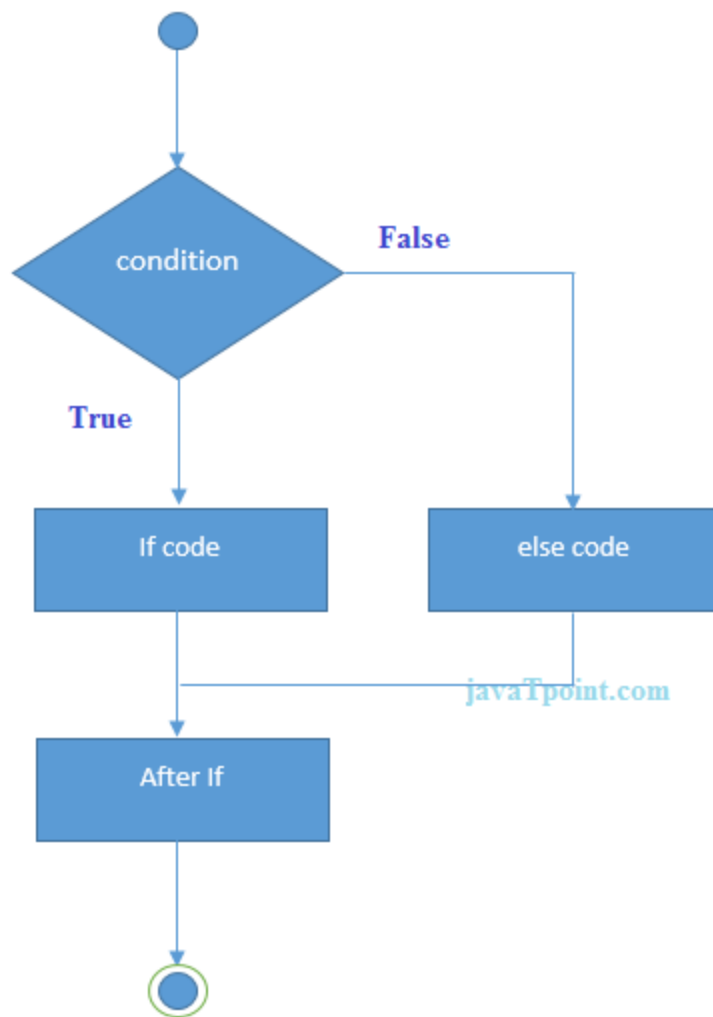
```
Enter three numbers?
12 23 34
34 is largest
```

If-else Statement

The if-else statement is used to perform two operations for a single condition. The if-else statement is an extension to the if statement using which, we can perform two different operations, i.e., one is for the correctness of that condition, and the other is for the incorrectness of the condition. Here, we must notice that if and else block cannot be executed simultaneously. Using if-else statement is always preferable since it always invokes an otherwise case with every if condition. The syntax of the if-else statement is given below.

```
1. if(expression){
2. //code to be executed if condition is true
3. }else{
4. //code to be executed if condition is false
5. }
```

Flowchart of the if-else statement in C



Let's see the simple example to check whether a number is even or odd using if-else statement in C language.

```
1. #include<stdio.h>
2. int main(){
3.     int number=0;
4.     printf("enter a number:");
5.     scanf("%d",&number);
6.     if(number%2==0){
7.         printf("%d is even number",number);
8.     }
```

```
9. else{
10. printf("%d is odd number",number);
11.}
12. return 0;
13.}
```

Output


```
enter a number:4
4 is even number
enter a number:5
5 is odd number
```

Program to check whether a person is eligible to vote or not.

```
1. #include <stdio.h>
2. int main()
3. {
4.     int age;
5.     printf("Enter your age?");
6.     scanf("%d",&age);
7.     if(age>=18)
8.     {
9.         printf("You are eligible to vote...");
10.    }
11.    else
12.    {
13.        printf("Sorry ... you can't vote");
14.    }
15.}
```

Output

```
Enter your age?18
You are eligible to vote...
Enter your age?13
Sorry ... you can't vote
```

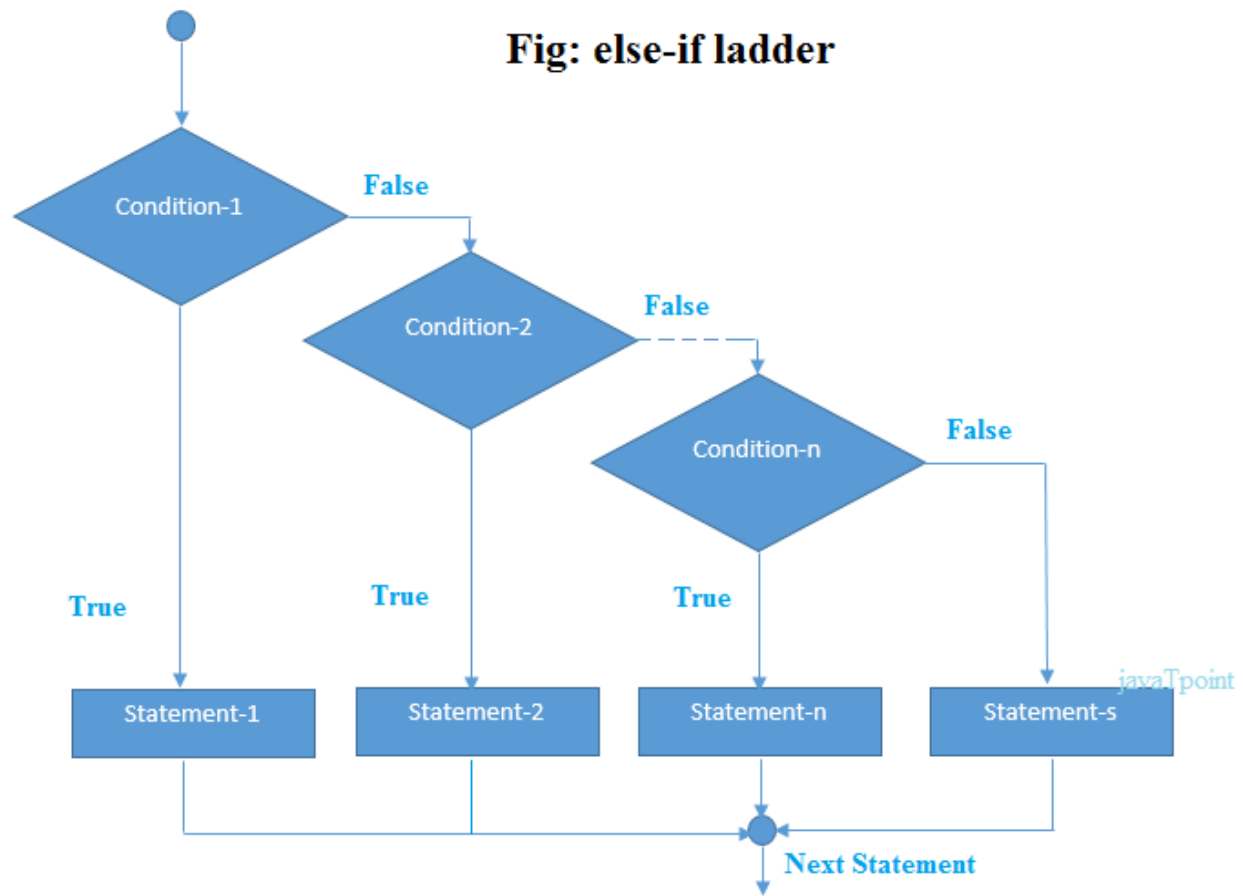


If else-if ladder Statement

The if-else-if ladder statement is an extension to the if-else statement. It is used in the scenario where there are multiple cases to be performed for different conditions. In if-else-if ladder statement, if a condition is true then the statements defined in the if block will be executed, otherwise if some other condition is true then the statements defined in the else-if block will be executed, at the last if none of the condition is true then the statements defined in the else block will be executed. There are multiple else-if blocks possible. It is similar to the switch case statement where the default is executed instead of else block if none of the cases is matched.

1. **if**(condition1){
2. //code to be executed if condition1 is true
3. }**else if**(condition2){
4. //code to be executed if condition2 is true
5. }
6. **else if**(condition3){
7. //code to be executed if condition3 is true
8. }
9. ...
10. **else**{
11. //code to be executed if all the conditions are false
12. }

Flowchart of else-if ladder statement in C



The example of an if-else-if statement in C language is given below.

```
1. #include<stdio.h>
2. int main(){
3. int number=0;
4. printf("enter a number:");
5. scanf("%d",&number);
6. if(number==10){
7. printf("number is equals to 10");
8. }
9. else if(number==50){
10. printf("number is equal to 50");
11. }
```

```
12. else if(number==100){
13. printf("number is equal to 100");
14.}
15. else{
16. printf("number is not equal to 10, 50 or 100");
17.}
18. return 0;
19.}
```

Output

```
enter a number:4
number is not equal to 10, 50 or 100
enter a number:50
number is equal to 50
```

Program to calculate the grade of the student according to the specified marks.

```
1. #include <stdio.h>
2. int main()
3. {
4.     int marks;
5.     printf("Enter your marks?");
6.     scanf("%d",&marks);
7.     if(marks > 85 && marks <= 100)
8.     {
9.         printf("Congrats ! you scored grade A ...");
10.    }
11.    else if (marks > 60 && marks <= 85)
12.    {
```

```

13.     printf("You scored grade B + ...");
14. }
15. else if (marks > 40 && marks <= 60)
16. {
17.     printf("You scored grade B ...");
18. }
19. else if (marks > 30 && marks <= 40)
20. {
21.     printf("You scored grade C ...");
22. }
23. else
24. {
25.     printf("Sorry you are fail ...");
26. }
27.}

```

```

#include<stdio.h>
int main(){
    int marks;
    printf("Enter the marks of a student:\n");
    scanf("%d",&marks);
    if(marks <=100 && marks >= 90)
        printf("Grade=A");
    else if(marks < 90 && marks>= 80)
        printf("Grade=B");
    else if(marks < 80 && marks >= 70)
        printf("Grade=C");
    else if(marks < 70 && marks >= 60)
        printf("Grade=D");
    else if(marks < 60 && marks > 50)
        printf("Grade=E");
    else if(marks == 50)
        printf("Grade=F");
}

```

```
else if(marks < 50 && marks >= 0)
    printf("Fail");
else
    printf("Enter a valid score between 0 and 100");
return 0;
}
```

C Switch Statement

The switch statement in C is an alternate to if-else-if ladder statement which allows us to execute multiple operations for the different possible values of a single variable called switch variable. Here, We can define various statements in the multiple cases for the different values of a single variable.

The syntax of switch statement in **c language** is given below:

1. **switch**(expression){
2. **case** value1:
3. *//code to be executed;*
4. **break;** *//optional*
5. **case** value2:
6. *//code to be executed;*
7. **break;** *//optional*
8.
- 9.
10. **default:**
11. code to be executed **if** all cases are not matched;
- 12.}

Rules for switch statement in C language

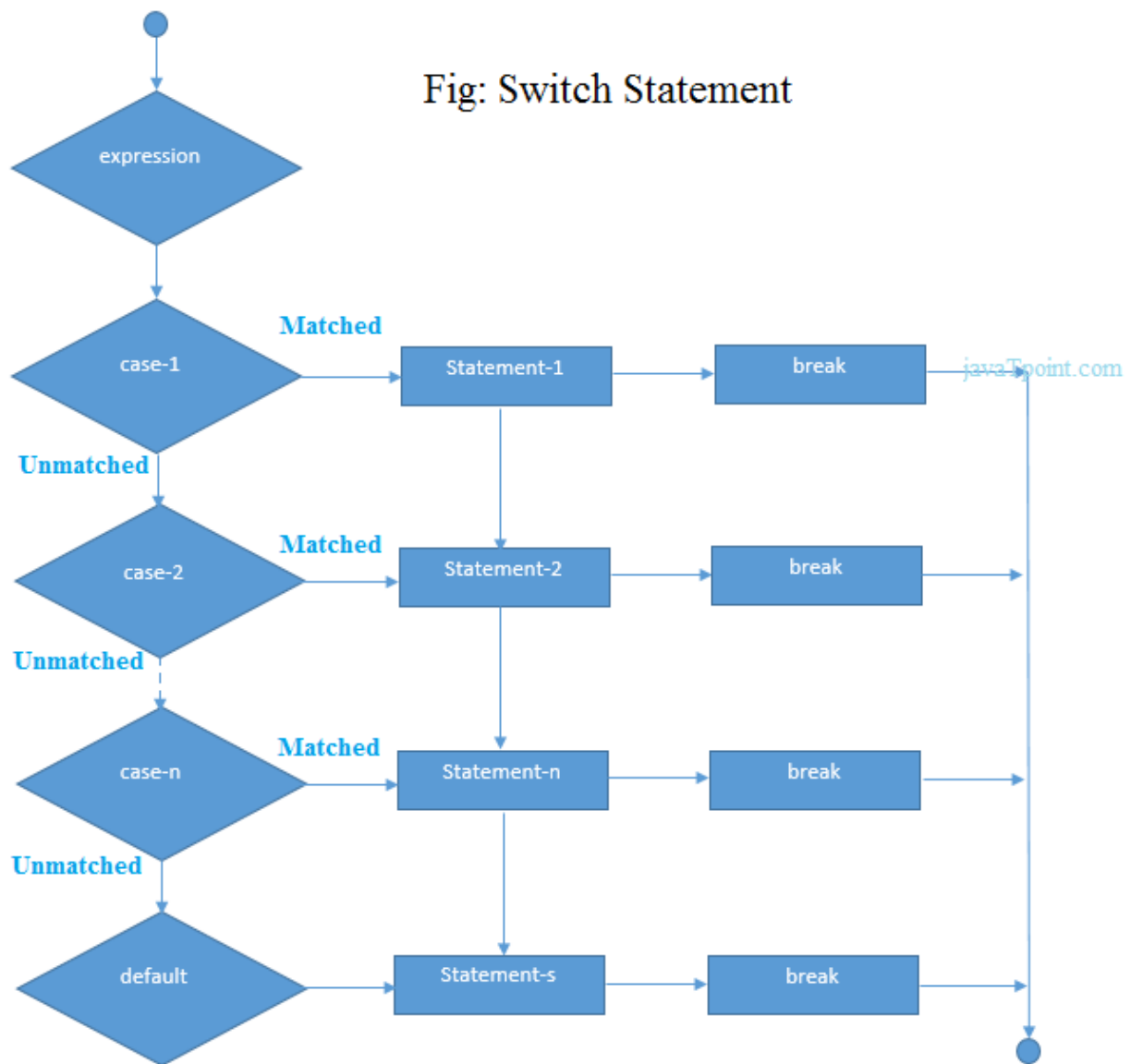
- 1) The *switch expression* must be of an integer or character type.
- 2) The *case value* must be an integer or character constant.
- 3) The *case value* can be used only inside the switch statement.
- 4) The *break statement* in switch case is not must. It is optional. If there is no break statement found in the case, all the cases will be executed present after the matched case. It is known as *fall through* the state of C switch statement.

Let's try to understand it by the examples. We are assuming that there are following variables.

1. **int** x,y,z;
2. **char** a,b;
3. **float** f;

Valid Switch	Invalid Switch	Valid Case	Invalid Case
switch(x)	switch(f)	case 3;	case 2.5;
switch(x>y)	switch(x+2.5)	case 'a';	case x;
switch(a+b-2)		case 1+2;	case x+2;
switch(func(x,y))		case 'x'>'y';	case 1,2,3;

Flowchart of switch statement in C



Functioning of switch case statement

First, the integer expression specified in the switch statement is evaluated. This value is then matched one by one with the constant values given in the different cases. If a match is found, then all the statements specified in that case are executed along with the all the cases present after that case including the default statement. No two cases can have similar values. If the matched case contains a break statement, then all the cases present after that will be skipped, and the control comes out of the switch. Otherwise, all the cases following the matched case will be executed.

Let's see a simple example of c language switch statement.

```
1. #include<stdio.h>
2. int main(){
3.     int number=0;
4.     printf("enter a number:");
5.     scanf("%d",&number);
6.     switch(number){
7.         case 10:
8.             printf("number is equals to 10");
9.             break;
10.        case 50:
11.            printf("number is equal to 50");
12.            break;
13.        case 100:
14.            printf("number is equal to 100");
15.            break;
16.        default:
17.            printf("number is not equal to 10, 50 or 100");
18.    }
19.    return 0;
20.}
```

Output

```
enter a number:4
number is not equal to 10, 50 or 100
```



```
enter a number:50
number is equal to 50
```



Switch case example 2

```
1. #include <stdio.h>
2. int main()
3. {
4.     int x = 10, y = 5;
5.     switch(x>y && x+y>0)
6.     {
7.         case 1:
8.             printf("hi");
9.             break;
10.        case 0:
11.            printf("bye");
12.            break;
13.        default:
14.            printf(" Hello bye ");
15.    }
16.
17.}
```

Output

hi

C Switch statement is fall-through

In C language, the switch statement is fall through; it means if you don't use a break statement in the switch case, all the cases after the matching case will be executed.

Let's try to understand the fall through state of switch statement by the example given below.

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



```
2. int main(){
3. int number=0;
4.
5. printf("enter a number:");
6. scanf("%d",&number);
7.
8. switch(number){
9. case 10:
10. printf("number is equal to 10\n");
11. case 50:
12. printf("number is equal to 50\n");
13. case 100:
14. printf("number is equal to 100\n");
15. default:
16. printf("number is not equal to 10, 50 or 100");
17. }
18. return 0;
19. }
```

Output

```
enter a number:10
number is equal to 10
number is equal to 50
number is equal to 100
number is not equal to 10, 50 or 100
```



Output

```
enter a number:50
number is equal to 50
number is equal to 100
```

number is not equal to 10, 50 or 100

Nested switch case statement

We can use as many switch statement as we want inside a switch statement. Such type of statements is called nested switch case statements. Consider the following example.

```
1. #include <stdio.h>
2. int main () {
3.
4.     int i = 10;
5.     int j = 20;
6.
7.     switch(i) {
8.
9.         case 10:
10.            printf("the value of i evaluated in outer switch: %d\n",i);
11.        case 20:
12.            switch(j) {
13.                case 20:
14.                    printf("The value of j evaluated in nested switch: %d\n",j);
15.            }
16.        }
17.
18.    printf("Exact value of i is : %d\n", i);
19.    printf("Exact value of j is : %d\n", j);
20.
21.    return 0;
22.}
```

Output

```
the value of i evaluated in outer switch: 10
The value of j evaluated in nested switch: 20
Exact value of i is : 10
Exact value of i is : 20
```

if-else vs switch

What is an if-else statement?

An if-else statement in C programming is a conditional statement that executes a different set of statements based on the condition that is true or false. The 'if' block will be executed only when the specified condition is true, and if the specified condition is false, then the else block will be executed.

Syntax of if-else statement

is given below:

1. **else**
2. { **if**(expression)
3. {
4. **// statements;**
5. }
- 6.
7. **// statements;**
8. }

What is a switch statement?

A switch statement

is a conditional statement used in C programming

to check the value of a variable and compare it with all the cases. If the value is matched with any case, then its corresponding statements will be executed. Each case has some name or number known as the identifier. The value entered by the user will be compared with all the cases until the case is found. If the value entered by the user is not matched with any case, then the default statement will be executed.

Syntax of the switch statement is given below:

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

Similarity b/w if-else and switch

Both the if-else and switch are the decision-making statements. Here, decision-making statements mean that the output of the expression will decide which statements are to be executed.

Differences b/w if-else and switch statement

The following are the differences between if-else and switch statement are:



- **Definition**

if-else

Based on the result of the expression in the 'if-else' statement, the block of statements will be executed. If the condition is true, then the 'if' block will be executed otherwise 'else' block will execute.

Switch statement

The switch statement contains multiple cases or choices. The user will decide the case, which is to execute.

- **Expression**

If-else

It can contain a single expression or multiple expressions for multiple choices. In this, an expression is evaluated based on the range of values or conditions. It checks both equality and logical expressions.

Switch

It contains only a single expression, and this expression is either a single integer object or a string object. It checks only equality expression.

- **Evaluation**

If-else

An if-else statement can evaluate almost all the types of data such as integer, floating-point, character, pointer, or Boolean.

Switch

A switch statement can evaluate either an integer or a character.

- **Sequence of Execution**

If-else

In the case of 'if-else' statement, either the 'if' block or the 'else' block will be executed based on the condition.

Switch

In the case of the 'switch' statement, one case after another will be executed until the **break** keyword is not found, or the default statement is executed.

- **Default Execution**

If-else

If the condition is not true within the 'if' statement, then by default, the else block statements will be executed.

Switch

If the expression specified within the **switch** statement is not matched with any of the cases, then the default statement, if defined, will be executed.

- **Values**

If-else

Values are based on the condition specified inside the 'if' statement. The value will decide either the 'if' or 'else' block is to be executed.

Switch

In this case, value is decided by the user. Based on the choice of the user, the case will be executed.

- **Use**

If-else

It evaluates a condition to be true or false.

Switch

A **switch** statement compares the value of the variable with multiple cases. If the value is matched with any of the cases, then the block of statements associated with this case will be executed.

- **Editing**

If-else

Editing in 'if-else' statement is not easy as if we remove the 'else' statement, then it will create the havoc.

Switch

Editing in **switch** statement is easier as compared to the 'if-else' statement. If we remove any of the cases from the switch, then it will not interrupt the execution of other cases. Therefore, we can say that the **switch** statement is easy to modify and maintain.

- **Speed**

If-else

If the choices are multiple, then the speed of the execution of 'if-else' statements is slow.

Switch

The case constants in the switch statement create a jump table at the compile time. This jump table chooses the path of the execution based on the value of the expression. If we have a multiple choice, then the execution of the switch statement will be much faster than the equivalent logic of 'if-else' statement.

Let's summarize the above differences in a tabular form.

	If-else	switch
Definition	Depending on the condition in the 'if' statement, 'if' and 'else' blocks are executed.	The user will decide which statement is to be executed.
Expression	It contains either logical or equality expression.	It contains a single expression which can be either a character or integer variable.
Evaluation	It evaluates all types of data, such as integer, floating-point, character or Boolean.	It evaluates either an integer, or character.
Sequence of execution	First, the condition is checked. If the condition is true then 'if' block is executed otherwise 'else' block	It executes one case after another till the break keyword is not found, or the default statement is executed.
Default execution	If the condition is not true, then by default, else block will be executed.	If the value does not match with any case, then by default, default statement is executed.

Editing	Editing is not easy in the 'if-else' statement.	Cases in a switch statement are easy to maintain and modify. Therefore, we can say that the removal or editing of any case will not interrupt the execution of other cases.
Speed	If there are multiple choices implemented through 'if-else', then the speed of the execution will be slow.	If we have multiple choices then the switch statement is the best option as the speed of the execution will be much higher than 'if-else'.

C Loops

The looping can be defined as repeating the same process multiple times until a specific condition satisfies. There are three types of loops used in the C language. In this part of the tutorial, we are going to learn all the aspects of C loops.

Why use loops in C language?

The looping simplifies the complex problems into the easy ones. It enables us to alter the flow of the program so that instead of writing the same code again and again, we can repeat the same code for a finite number of times. For example, if we need to print the first 10 natural numbers then, instead of using the printf statement 10 times, we can print inside a loop which runs up to 10 iterations.

Advantage of loops in C

- 1) It provides code reusability.
- 2) Using loops, we do not need to write the same code again and again.

3) Using loops, we can traverse over the elements of data structures (array or linked lists).

Types of C Loops

There are three types of loops in C language that is given below:

1. do while
2. while
3. for

do-while loop in C

The do-while loop continues until a given condition satisfies. It is also called post tested loop. It is used when it is necessary to execute the loop at least once (mostly menu driven programs).

The syntax of do-while loop in c language is given below:

1. **do**{
2. *//code to be executed*
3. **}while**(condition);

Flowchart and Example of do-while loop

while loop in C

The while loop in c is to be used in the scenario where we don't know the number of iterations in advance. The block of statements is executed in the while loop until the condition specified in the while loop is satisfied. It is also called a pre-tested loop.

The syntax of while loop in c language is given below:

1. **while**(condition){
2. *//code to be executed*
3. }

Flowchart and Example of while loop

for loop in C

The for loop is used in the case where we need to execute some part of the code until the given condition is satisfied. The for loop is also called as a pre-tested loop. It is better to use for loop if the number of iteration is known in advance.

The syntax of for loop in c language is given below:

1. **for**(initialization;condition;incr/decr){
2. *//code to be executed*
3. }

do while loop in C

The do while loop is a post tested loop. Using the do-while loop, we can repeat the execution of several parts of the statements. The do-while loop is mainly used in the case where we need to execute the loop at least once. The do-while loop is mostly used in menu-driven programs where the termination condition depends upon the end user.

do while loop syntax

The syntax of the C language do-while loop is given below:

1. **do**{
2. *//code to be executed*
3. }**while**(condition);

Example 1

1. **#include**<stdio.h>
2. **#include**<stdlib.h>
3. **void** main ()
4. {
5. **char** c;

```
6.  int choice,dummy;
7.  do{
8.  printf("\n1. Print Hello\n2. Print Javatpoint\n3. Exit\n");
9.  scanf("%d",&choice);
10. switch(choice)
11. {
12.     case 1 :
13.         printf("Hello");
14.         break;
15.     case 2:
16.         printf("Javatpoint");
17.         break;
18.     case 3:
19.         exit(0);
20.         break;
21.     default:
22.         printf("please enter valid choice");
23. }
24. printf("do you want to enter more?");
25. scanf("%d",&dummy);
26. scanf("%c",&c);
27. }while(c=='y');
28. }
```

Output

```
1. Print Hello
2. Print Javatpoint
3. Exit
1
Hello
do you want to enter more?
```

y

1. Print Hello
2. Print Javatpoint
3. Exit

2

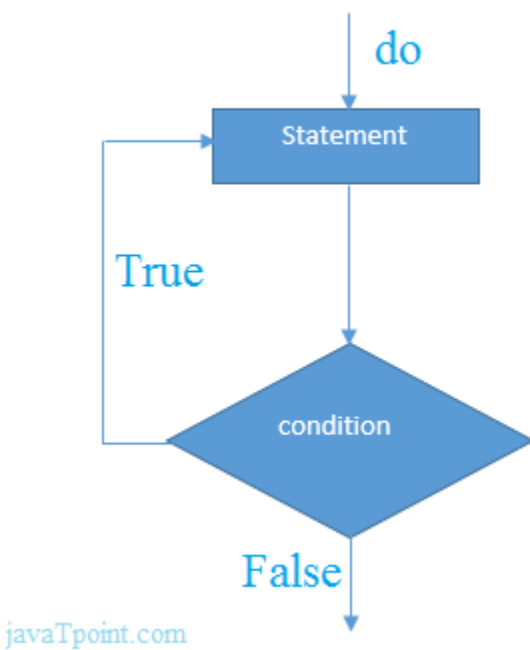
Javatpoint

do you want to enter more?

n



Flowchart of do while loop



do while example

There is given the simple program of c language do while loop where we are printing the table of 1.

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

```
4. do{
5. printf("%d \n",i);
6. i++;
7. }while(i<=10);
8. return 0;
9. }
```

Output

```
1
2
3
4
5
6
7
8
9
10
```

Program to print table for the given number using do while loop

```
1. #include<stdio.h>
2. int main(){
3. int i=1,number=0;
4. printf("Enter a number: ");
5. scanf("%d",&number);
6. do{
7. printf("%d \n",(number*i));
8. i++;
9. }while(i<=10);
10. return 0;
11.}
```

Output

Enter a number: 5

5
10
15
20
25
30
35
40
45
50



Enter a number: 10

10
20
30
40
50
60
70
80
90
100



Infinitive do while loop

The do-while loop will run infinite times if we pass any non-zero value as the conditional expression.

1. `do{`
2. `//statement`
3. `}while(1);`

while loop in C

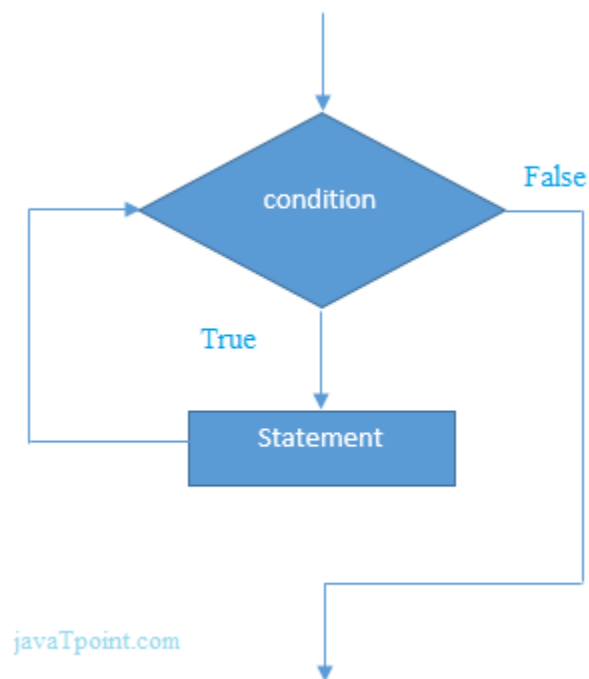
While loop is also known as a pre-tested loop. In general, a while loop allows a part of the code to be executed multiple times depending upon a given boolean condition. It can be viewed as a repeating if statement. The while loop is mostly used in the case where the number of iterations is not known in advance.

Syntax of while loop in C language

The syntax of while loop in c language is given below:

1. **while**(condition){
2. *//code to be executed*
3. }

Flowchart of while loop in C



Example of the while loop in C language

Let's see the simple program of while loop that prints table of 1.

1. **#include<stdio.h>**


```
2. int main(){
3. int i=1;
4. while(i<=10){
5. printf("%d \n",i);
6. i++;
7. }
8. return 0;
9. }
```

Output

```
1
2
3
4
5
6
7
8
9
10
```

Program to print table for the given number using while loop in C

```
1. #include<stdio.h>
2. int main(){
3. int i=1,number=0,b=9;
4. printf("Enter a number: ");
5. scanf("%d",&number);
6. while(i<=10){
7. printf("%d \n",(number*i));
```

```
8. i++;  
9. }  
10. return 0;  
11.}
```

Output

Enter a number: 50

50
100
150
200
250
300
350
400
450
500



Enter a number: 100

100
200
300
400
500
600
700
800
900
1000



Properties of while loop

- A conditional expression is used to check the condition. The statements defined inside the while loop will repeatedly execute until the given condition fails.
- The condition will be true if it returns 0. The condition will be false if it returns any non-zero number.
- In while loop, the condition expression is compulsory.
- Running a while loop without a body is possible.
- We can have more than one conditional expression in while loop.
- If the loop body contains only one statement, then the braces are optional.

Example 1

```
1. #include<stdio.h>
2. void main ()
3. {
4.     int j = 1;
5.     while(j+=2,j<=10)
6.     {
7.         printf("%d ",j);
8.     }
9.     printf("%d",j);
10.}
```

Output

3 5 7 9 11



Example 2

```
1. #include<stdio.h>
2. void main ()
3. {
```

```
4.  while()
5.  {
6.      printf("hello Javatpoint");
7.  }
8. }
```

Output

compile time error: while loop can't be empty



Example 3

```
1. #include<stdio.h>
2. void main ()
3. {
4.     int x = 10, y = 2;
5.     while(x+y-1)
6.     {
7.         printf("%d %d",x--,y--);
8.     }
9. }
```

Output

infinite loop



Infinite while loop in C

If the expression passed in while loop results in any non-zero value then the loop will run the infinite number of times.

```
1. while(1){
2.     //statement
```

3. }

for loop in C

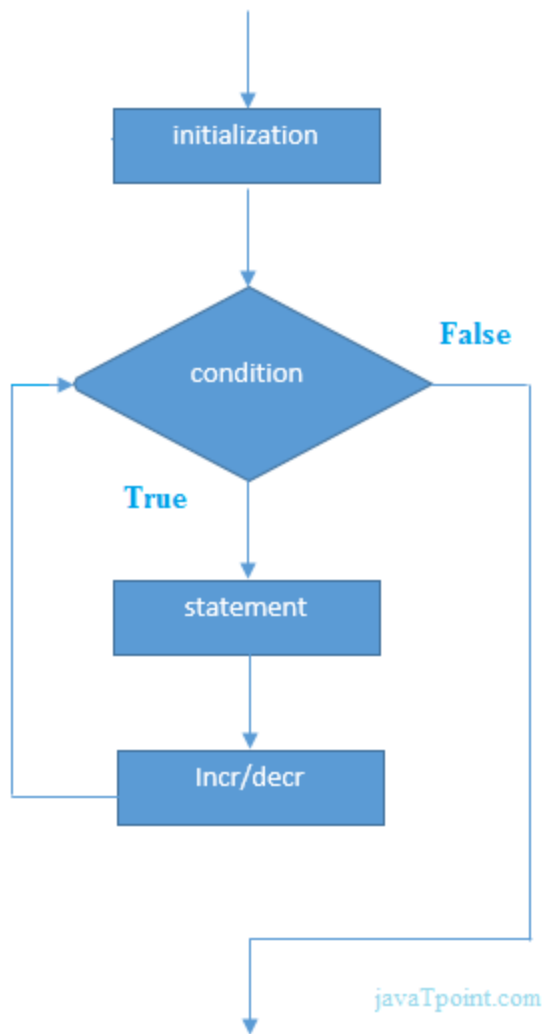
The **for loop in C language** is used to iterate the statements or a part of the program several times. It is frequently used to traverse the data structures like the array and linked list.

Syntax of for loop in C

The syntax of for loop in c language is given below:

1. **for**(Expression 1; Expression 2; Expression 3){
2. *//code to be executed*
3. }

Flowchart of for loop in C



C for loop Examples

Let's see the simple program of for loop that prints table of 1.

1. `#include<stdio.h>`
2. `int main(){`
3. `int i=0;`
4. `for(i=1;i<=10;i++){`
5. `printf("%d \n",i);`
6. `}`

7. `return 0;`
8. `}`

Output

1
2
3
4
5
6
7
8
9
10

C Program: Print table for the given number using C for loop

1. `#include<stdio.h>`
2. `int main(){`
3. `int i=1,number=0;`
4. `printf("Enter a number: ");`
5. `scanf("%d",&number);`
6. `for(i=1;i<=10;i++){`
7. `printf("%d \n",(number*i));`
8. `}`
9. `return 0;`
10. `}`

Output

Enter a number: 2

2
4
6
8
10
12
14
16
18
20



Enter a number: 1000

1000
2000
3000
4000
5000
6000
7000
8000
9000
10000



Properties of Expression 1

- The expression represents the initialization of the loop variable.
- We can initialize more than one variable in Expression 1.
- Expression 1 is optional.
- In C, we can not declare the variables in Expression 1. However, It can be an exception in some compilers.

Example 1

1. `#include <stdio.h>`


```
2. int main()
3. {
4.     int a,b,c;
5.     for(a=0,b=12,c=23;a<2;a++)
6.     {
7.         printf("%d ",a+b+c);
8.     }
9. }
```

Output

35 36

Example 2

```
1. #include <stdio.h>
2. int main()
3. {
4.     int i=1;
5.     for(;i<5;i++)
6.     {
7.         printf("%d ",i);
8.     }
9. }
```

Output

1 2 3 4

Properties of Expression 2

- Expression 2 is a conditional expression. It checks for a specific condition to be satisfied. If it is not, the loop is terminated.
- Expression 2 can have more than one condition. However, the loop will iterate until the last condition becomes false. Other conditions will be treated as statements.
- Expression 2 is optional.
- Expression 2 can perform the task of expression 1 and expression 3. That is, we can initialize the variable as well as update the loop variable in expression 2 itself.
- We can pass zero or non-zero value in expression 2. However, in C, any non-zero value is true, and zero is false by default.

Example 1

```

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

output

0 1 2 3 4



Example 2

```

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

```
2. int main()
3. {
4.     int i,j,k;
5.     for(i=0,j=0,k=0;i<4,k<8,j<10;i++)
6.     {
7.         printf("%d %d %d\n",i,j,k);
8.         j+=2;
9.         k+=3;
10.    }
11.}
```

Output

```
0 0 0
1 2 3
2 4 6
3 6 9
4 8 12
```



Example 3

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

Output

infinite loop

Properties of Expression 3

- Expression 3 is used to update the loop variable.
- We can update more than one variable at the same time.
- Expression 3 is optional.

Example 1

```
1. #include<stdio.h>
2. void main ()
3. {
4.     int i=0,j=2;
5.     for(i = 0;i<5;i++,j=j+2)
6.     {
7.         printf("%d %d\n",i,j);
8.     }
9. }
```

Output

```
0 2
1 4
2 6
3 8
4 10
```

Loop body

The braces {} are used to define the scope of the loop. However, if the loop contains only one statement, then we don't need to use braces. A loop without a body is possible.

The braces work as a block separator, i.e., the value variable declared inside for loop is valid only for that block and not outside. Consider the following example.

```
1. #include<stdio.h>
2. void main ()
3. {
4.     int i;
5.     for(i=0;i<10;i++)
6.     {
7.         int i = 20;
8.         printf("%d ",i);
9.     }
10.}
```

Output

20 20 20 20 20 20 20 20 20 20



Infinitive for loop in C

To make a for loop infinite, we need not give any expression in the syntax. Instead of that, we need to provide two semicolons to validate the syntax of the for loop. This will work as an infinite for loop.

```
1. #include<stdio.h>
2. void main ()
3. {
4.     for(;;)
5.     {
6.         printf("welcome to javatpoint");
7.     }
8. }
```

If you run this program, you will see above statement infinite times.

C break statement

The break is a keyword in C which is used to bring the program control out of the loop. The break statement is used inside loops or switch statement. The break statement breaks the loop one by one, i.e., in the case of nested loops, it breaks the inner loop first and then proceeds to outer loops. The break statement in C can be used in the following two scenarios:

1. With switch case
2. With loop

Syntax:

1. `//loop or switch case`
2. `break;`

Flowchart of break in c

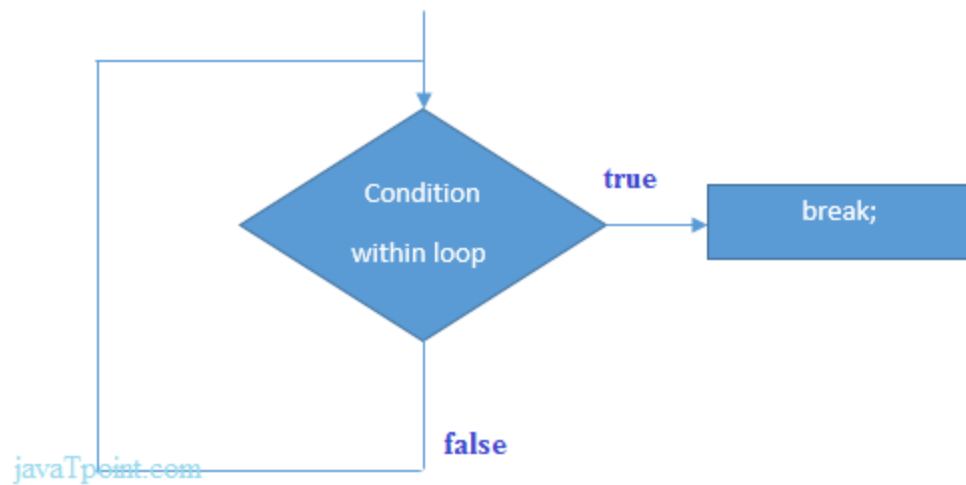


Figure: Flowchart of break statement

Example

```
1. #include<stdio.h>
2. #include<stdlib.h>
3. void main ()
4. {
5.     int i;
6.     for(i = 0; i<10; i++)
7.     {
8.         printf("%d ",i);
9.         if(i == 5)
10.            break;
11.     }
12.     printf("came outside of loop i = %d",i);
13.
14. }
```

Output

0 1 2 3 4 5 came outside of loop i = 5

Example of C break statement with switch case

[Click here to see the example of C break with the switch statement.](#)

C break statement with the nested loop

In such case, it breaks only the inner loop, but not outer loop.

```
1. #include<stdio.h>
2. int main(){
3.     int i=1,j=1;//initializing a local variable
4.     for(i=1;i<=3;i++){
5.         for(j=1;j<=3;j++){
6.             printf("%d &d\n",i,j);
7.             if(i==2 && j==2){
8.                 break;//will break loop of j only
9.             }
10.        }//end of for loop
11.    return 0;
12.}
```

Output

1 1

1 2

1 3

2 1

2 2

3 1

3 2

3 3

As you can see the output on the console, 2 3 is not printed because there is a break statement after printing i==2 and j==2. But 3 1, 3 2 and 3 3 are printed because the break statement is used to break the inner loop only.

break statement with while loop

Consider the following example to use break statement inside while loop.

```
1. #include<stdio.h>
2. void main ()
3. {
4.     int i = 0;
5.     while(1)
6.     {
7.         printf("%d ",i);
8.         i++;
9.         if(i == 10)
10.            break;
11.    }
12.    printf("came out of while loop");
13.}
```

Output

0 1 2 3 4 5 6 7 8 9 came out of while loop

break statement with do-while loop

Consider the following example to use the break statement with a do-while loop.

```
1. #include<stdio.h>
2. void main ()
3. {
4.     int n=2,i,choice;
5.     do
6.     {
7.         i=1;
8.         while(i<=10)
9.         {
10.            printf("%d X %d = %d\n",n,i,n*i);
11.            i++;
12.        }
13.        printf("do you want to continue with the table of %d , enter any non-zero value  
to continue.",n+1);
14.        scanf("%d",&choice);
15.        if(choice == 0)
16.        {
17.            break;
18.        }
19.        n++;
20.    }while(1);
21.}
```

Next → ← Prev

C continue statement

The continue statement in C language is used to bring the program control to the beginning of the loop. The continue statement skips some lines of code inside the loop and continues with the next iteration. It is mainly used for a condition so that we can skip some code for a particular condition.

Syntax:

```
//loop statements
```

```
continue;
```

```
//some lines of the code which is to be skipped
```

Continue statement example 1

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

```
void main ()
```

```
{
```

```
    int i = 0;
```

```
    while(i!=10)
```

```
    {
```

```
        printf("%d", i);
```

```
        continue;
```

```
        i++;
```

```
    }
```

```
}
```

Output

infinite loop

Continue statement example 2

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

```
int main(){
```

```
int i=1;//initializing a local variable
```

```
//starting a loop from 1 to 10
```

```
for(i=1;i<=10;i++){
```

```
if(i==5){//if value of i is equal to 5, it will continue the loop
```

```
continue;
```

```
}
```

```
printf("%d \n",i);
```

```
}//end of for loop
```

```
return 0;
```

```
}
```

Output

1

2

3

4

6
7
8
9
10

As you can see, 5 is not printed on the console because loop is continued at `i==5`.

C continue statement with inner loop

In such case, C continue statement continues only inner loop, but not outer loop.

```
#include<stdio.h>

int main(){
    int i=1,j=1;//initializing a local variable
    for(i=1;i<=3;i++){
        for(j=1;j<=3;j++){
            if(i==2 && j==2){
                continue;//will continue loop of j only
            }
            printf("%d %d\n",i,j);
        }
    }
}
```

```
return 0;

}
```

C goto statement

The goto statement is known as jump statement in C. As the name suggests, goto is used to transfer the program control to a predefined label. The goto statement can be used to repeat some part of the code for a particular condition. It can also be used to break the multiple loops which can't be done by using a single break statement. However, using goto is avoided these days since it makes the program less readable and complicated.

Syntax:

1. label:
2. `//some part of the code;`
3. `goto label;`

goto example

Let's see a simple example to use goto statement in C language.

1. `#include <stdio.h>`
2. `int main()`
3. `{`
4. `int num,i=1;`
5. `printf("Enter the number whose table you want to print?");`
6. `scanf("%d",&num);`
7. `table:`
8. `printf("%d x %d = %d\n",num,i,num*i);`
9. `i++;`
10. `if(i<=10)`

```
11. goto table;
12.}
```

Output:

Enter the number whose table you want to print?10

10 x 1 = 10

10 x 2 = 20

10 x 3 = 30

10 x 4 = 40

10 x 5 = 50

10 x 6 = 60

10 x 7 = 70

10 x 8 = 80

10 x 9 = 90

10 x 10 = 100



When should we use goto?

The only condition in which using goto is preferable is when we need to break the multiple loops using a single statement at the same time. Consider the following example.

```
1. #include <stdio.h>
2. int main()
3. {
```

```
4.  int i, j, k;
5.  for(i=0;i<10;i++)
6.  {
7.      for(j=0;j<5;j++)
8.      {
9.          for(k=0;k<3;k++)
10.         {
11.             printf("%d %d %d\n",i,j,k);
12.             if(j == 3)
13.             {
14.                 goto out;
15.             }
16.         }
17.     }
18. }
19. out:
20. printf("came out of the loop");
21. }
```

Type Casting in C

Typecasting allows us to convert one data type into other. In C language, we use cast operator for typecasting which is denoted by (type).

Syntax:

```
1. (type)value;
```

Note: It is always recommended to convert the lower value to higher for avoiding data loss.

Without Type Casting:

1. `int f= 9/4;`
2. `printf("f : %d\n", f);`//Output: 2

With Type Casting:

1. `float f=(float) 9/4;`
2. `printf("f : %f\n", f);`//Output: 2.250000

Type Casting example

Let's see a simple example to cast int value into the float.

1. `#include<stdio.h>`
2. `int main(){`
3. `float f= (float)9/4;`
4. `printf("f : %f\n", f);`
5. `return 0;`
6. `}`

Output:

f : 2.250000

C Functions

In c, we can divide a large program into the basic building blocks known as function. The function contains the set of programming statements enclosed by {}. A function can be called multiple times to provide reusability and modularity to the C program. In other words, we can say that the collection of functions creates a program. The function is also known as *procedure* or *subroutine* in other programming languages.

Advantage of functions in C

There are the following advantages of C functions.

- By using functions, we can avoid rewriting same logic/code again and again in a program.
- We can call C functions any number of times in a program and from any place in a program.
- We can track a large C program easily when it is divided into multiple functions.
- Reusability is the main achievement of C functions.
- However, Function calling is always a overhead in a C program.

Function Aspects

There are three aspects of a C function.

- **Function declaration** A function must be declared globally in a c program to tell the compiler about the function name, function parameters, and return type.
- **Function call** Function can be called from anywhere in the program. The parameter list must not differ in function calling and function declaration. We must pass the same number of functions as it is declared in the function declaration.
- **Function definition** It contains the actual statements which are to be executed. It is the most important aspect to which the control comes when the function is

called. Here, we must notice that only one value can be returned from the function.

S N	C function aspects	Syntax
1	Function declaration	return_type function_name (argument list);
2	Function call	function_name (argument_list)
3	Function definition	return_type function_name (argument list) {function body;}

The syntax of creating function in c language is given below:

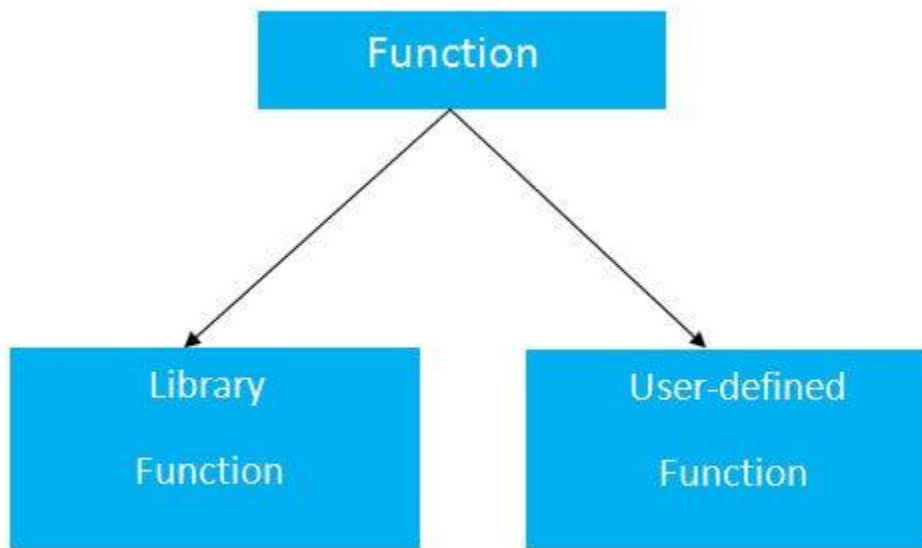
1. return_type function_name(data_type parameter...){
2. `//code to be executed`
3. }

Types of Functions

There are two types of functions in C programming:

1. **Library Functions:** are the functions which are declared in the C header files such as scanf(), printf(), gets(), puts(), ceil(), floor() etc.

2. **User-defined functions:** are the functions which are created by the C programmer, so that he/she can use it many times. It reduces the complexity of a big program and optimizes the code.



Assignment:

1. Write a c program which demonstrates usage of functions

No. of fun 5

Return Value

A C function may or may not return a value from the function. If you don't have to return any value from the function, use void for the return type.

Let's see a simple example of C function that doesn't return any value from the function.

Example without return value:

1. `void hello(){`

2. `printf("hello c");`
3. `}`

If you want to return any value from the function, you need to use any data type such as int, long, char, etc. The return type depends on the value to be returned from the function.

Let's see a simple example of C function that returns int value from the function.

Example with return value:

1. `int get(){`
2. `return 10;`
3. `}`

In the above example, we have to return 10 as a value, so the return type is int. If you want to return floating-point value (e.g., 10.2, 3.1, 54.5, etc), you need to use float as the return type of the method.

1. `float get(){`
2. `return 10.2;`
3. `}`

Now, you need to call the function, to get the value of the function.

Different aspects of function calling

A function may or may not accept any argument. It may or may not return any value. Based on these facts, There are four different aspects of function calls.

- function without arguments and without return value
- function without arguments and with return value
- function with arguments and without return value
- function with arguments and with return value

Example for Function without argument and return value

Example 1

```
1. #include<stdio.h>
2. void printName();
3. void main ()
4. {
5.     printf("Hello ");
6.     printName();
7. }
8. void printName()
9. {
10.    printf("Javatpoint");
11.}
```

Output

Hello Javatpoint



Example 2

```
1. #include<stdio.h>
2. void sum();
3. void main()
4. {
5.     printf("\nGoing to calculate the sum of two numbers:");
6.     sum();
7. }
8. void sum()
9. {
```

```
10. int a,b;
11. printf("\nEnter two numbers");
12. scanf("%d %d",&a,&b);
13. printf("The sum is %d",a+b);
14.}
```

Output

Going to calculate the sum of two numbers:

Enter two numbers 10

24

The sum is 34

Example for Function without argument and with return value

Example 1

```
1. #include<stdio.h>
2. int sum();
3. void main()
4. {
5.     int result;
6.     printf("\nGoing to calculate the sum of two numbers:");
7.     result = sum();
8.     printf("%d",result);
9. }
10.int sum()
```

```
11.{
12.  int a,b;
13.  printf("\nEnter two numbers");
14.  scanf("%d %d",&a,&b);
15.  return a+b;
16.}
```

Output

Going to calculate the sum of two numbers:

Enter two numbers 10

24

The sum is 34



Example 2: program to calculate the area of the square

```
1. #include<stdio.h>
2. int sum();
3. void main()
4. {
5.     printf("Going to calculate the area of the square\n");
6.     float area = square();
7.     printf("The area of the square: %f\n",area);
8. }
9. int square()
10.{
11.    float side;
```



```
12. printf("Enter the length of the side in meters: ");
13. scanf("%f",&side);
14. return side * side;
15.}
```

Output

Going to calculate the area of the square

Enter the length of the side in meters: 10

The area of the square: 100.000000

Example for Function with argument and without return value

Example 1

```
1. #include<stdio.h>
2. void sum(int, int);
3. void main()
4. {
5.     int a,b,result;
6.     printf("\nGoing to calculate the sum of two numbers:");
7.     printf("\nEnter two numbers:");
8.     scanf("%d %d",&a,&b);
9.     sum(a,b);
10.}
11. void sum(int a, int b)
12.{
13.    printf("\nThe sum is %d",a+b);
14.}
```

Output

Going to calculate the sum of two numbers:

Enter two numbers 10

24

The sum is 34

Example 2: program to calculate the average of five numbers.

```
1. #include<stdio.h>
2. void average(int, int, int, int, int);
3. void main()
4. {
5.     int a,b,c,d,e;
6.     printf("\nGoing to calculate the average of five numbers:");
7.     printf("\nEnter five numbers:");
8.     scanf("%d %d %d %d %d",&a,&b,&c,&d,&e);
9.     average(a,b,c,d,e);
10.}
11.void average(int a, int b, int c, int d, int e)
12.{
13.    float avg;
14.    avg = (a+b+c+d+e)/5;
15.    printf("The average of given five numbers : %f",avg);
16.}
```

Output

Going to calculate the average of five numbers:

Enter five numbers:10

20

30

40

50

The average of given five numbers : 30.000000

Example for Function with argument and with return value

Example 1

```
1. #include<stdio.h>
2. int sum(int, int);
3. void main()
4. {
5.     int a,b,result;
6.     printf("\nGoing to calculate the sum of two numbers:");
7.     printf("\nEnter two numbers:");
8.     scanf("%d %d",&a,&b);
9.     result = sum(a,b);
10.    printf("\nThe sum is : %d",result);
11.}
12.int sum(int a, int b)
13.{
14.    return a+b;
15.}
```

Output

Going to calculate the sum of two numbers:

Enter two numbers:10

20

The sum is : 30

Example 2: Program to check whether a number is even or odd

```
1. #include<stdio.h>
2. int even_odd(int);
3. void main()
4. {
5.     int n,flag=0;
6.     printf("\nGoing to check whether a number is even or odd");
7.     printf("\nEnter the number: ");
8.     scanf("%d",&n);
9.     flag = even_odd(n);
10.    if(flag == 0)
11.    {
12.        printf("\nThe number is odd");
13.    }
14.    else
15.    {
16.        printf("\nThe number is even");
17.    }
18.}
19.int even_odd(int n)
20.{
```

```
21.  if(n%2 == 0)
22.  {
23.      return 1;
24.  }
25.  else
26.  {
27.      return 0;
28.  }
29. }
```

Output

Going to check whether a number is even or odd

Enter the number: 100

The number is even



C Library Functions

Library functions are the inbuilt function in C that are grouped and placed at a common place called the library. Such functions are used to perform some specific operations. For example, printf is a library function used to print on the console. The library functions are created by the designers of compilers. All C standard library functions are defined inside the different header files saved with the extension **.h**. We need to include these header files in our program to make use of the library functions defined in such header files. For example, To use the library functions such as printf/scanf we need to include `stdio.h` in our program which is a header file that contains all the library functions regarding standard input/output.

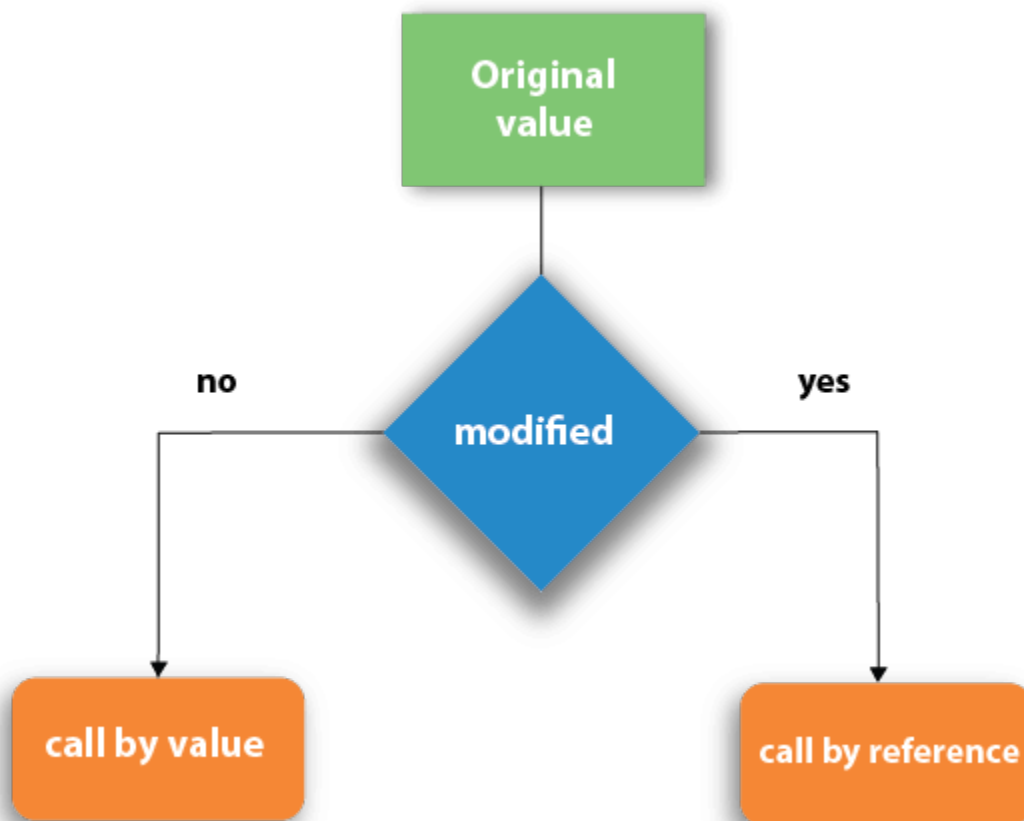
The list of mostly used header files is given in the following table.

S N	Header file	Description
1	stdio.h	This is a standard input/output header file. It contains all the library functions regarding standard input/output.
2	conio.h	This is a console input/output header file.
3	string.h	It contains all string related library functions like gets(), puts(),etc.
4	stdlib.h	This header file contains all the general library functions like malloc(), calloc(), exit(), etc.
5	math.h	This header file contains all the math operations related functions like sqrt(), pow(), etc.
6	time.h	This header file contains all the time-related functions.
7	ctype.h	This header file contains all character handling functions.
8	stdarg.h	Variable argument functions are defined in this header file.

9	signal.h	All the signal handling functions are defined in this header file.
10	setjmp.h	This file contains all the jump functions.
11	locale.h	This file contains locale functions.
12	errno.h	This file contains error handling functions.
13	assert.h	This file contains diagnostics functions.

Call by value and Call by reference in C

There are two methods to pass the data into the function in C language, i.e., *call by value* and *call by reference*.



Let's understand call by value and call by reference in c language one by one.

Call by value in C

- In call by value method, the value of the actual parameters is copied into the formal parameters. In other words, we can say that the value of the variable is used in the function call in the call by value method.
- In call by value method, we can not modify the value of the actual parameter by the formal parameter.
- In call by value, different memory is allocated for actual and formal parameters since the value of the actual parameter is copied into the formal parameter.
- The actual parameter is the argument which is used in the function call whereas formal parameter is the argument which is used in the function definition.

Let's try to understand the concept of call by value in c language by the example given below:

```

1. #include<stdio.h>
2. void change(int num) {
3.     printf("Before adding value inside function num=%d \n",num);
4.     num=num+100;
5.     printf("After adding value inside function num=%d \n", num);
6. }
7. int main() {
8.     int x=100;
9.     printf("Before function call x=%d \n", x);
10.    change(x);//passing value in function
11.    printf("After function call x=%d \n", x);
12.    return 0;
13.}

```

Output

Before function call x=100

Before adding value inside function num=100

After adding value inside function num=200

After function call x=100

Call by Value Example: Swapping the values of the two variables

```
1. #include <stdio.h>
2. void swap(int , int); //prototype of the function
3. int main()
4. {
5.     int a = 10;
6.     int b = 20;
7.     printf("Before swapping the values in main a = %d, b = %d\n",a,b); // printing the
        value of a and b in main
8.     swap(a,b);
9.     printf("After swapping values in main a = %d, b = %d\n",a,b); // The value of
        actual parameters do not change by changing the formal parameters in call by
        value, a = 10, b = 20
10.}
11. void swap (int a, int b)
12.{
13.     int temp;
14.     temp = a;
15.     a=b;
16.     b=temp;
17.     printf("After swapping values in function a = %d, b = %d\n",a,b); // Formal
        parameters, a = 20, b = 10
18.}
```

Output

Before swapping the values in main a = 10, b = 20

After swapping values in function a = 20, b = 10

After swapping values in main a = 10, b = 20

Call by reference in C

- In call by reference, the address of the variable is passed into the function call as the actual parameter.
- The value of the actual parameters can be modified by changing the formal parameters since the address of the actual parameters is passed.
- In call by reference, the memory allocation is similar for both formal parameters and actual parameters. All the operations in the function are performed on the value stored at the address of the actual parameters, and the modified value gets stored at the same address.

Consider the following example for the call by reference.

1. `#include<stdio.h>`
2. `void change(int *num) {`
3. `printf("Before adding value inside function num=%d \n",*num);`
4. `(*num) += 100;`
5. `printf("After adding value inside function num=%d \n", *num);`
6. `}`
7. `int main() {`
8. `int x=100;`
9. `printf("Before function call x=%d \n", x);`
10. `change(&x);//passing reference in function`

```
11. printf("After function call x=%d \n", x);
12. return 0;
13.}
```

Output

Before function call x=100

Before adding value inside function num=100

After adding value inside function num=200

After function call x=200



Call by reference Example: Swapping the values of the two variables

```
1. #include <stdio.h>
2. void swap(int *, int *); //prototype of the function
3. int main()
4. {
5.     int a = 10;
6.     int b = 20;
7.     printf("Before swapping the values in main a = %d, b = %d\n",a,b); // printing the
        value of a and b in main
8.     swap(&a,&b);
9.     printf("After swapping values in main a = %d, b = %d\n",a,b); // The values of
        actual parameters do change in call by reference, a = 10, b = 20
10.}
11. void swap (int *a, int *b)
12.{
13.     int temp;
14.     temp = *a;
```

```

15.  *a=*b;
16.  *b=temp;
17.  printf("After swapping values in function a = %d, b = %d\n",*a,*b); // Formal
    parameters, a = 20, b = 10
18.}

```

Output

Before swapping the values in main a = 10, b = 20

After swapping values in function a = 20, b = 10

After swapping values in main a = 20, b = 10

Difference between call by value and call by reference in c

No.	Call by value	Call by reference
1	A copy of the value is passed into the function	An address of value is passed into the function
2	Changes made inside the function is limited to the function only. The values of the actual parameters do not	Changes made inside the function validate outside of the function also. The values of the actual parameters do

	change by changing the formal parameters.	change by changing the formal parameters.
3	Actual and formal arguments are created at the different memory location	Actual and formal arguments are created at the same memory location

Recursion in C

Recursion is the process which comes into existence when a function calls a copy of itself to work on a smaller problem. Any function which calls itself is called recursive function, and such function calls are called recursive calls. Recursion involves several numbers of recursive calls. However, it is important to impose a termination condition of recursion. Recursion code is shorter than iterative code however it is difficult to understand.

Recursion cannot be applied to all the problem, but it is more useful for the tasks that can be defined in terms of similar subtasks. For Example, recursion may be applied to sorting, searching, and traversal problems.

Generally, iterative solutions are more efficient than recursion since function call is always overhead. Any problem that can be solved recursively, can also be solved iteratively. However, some problems are best suited to be solved by the recursion, for example, tower of Hanoi, Fibonacci series, factorial finding, etc.

In the following example, recursion is used to calculate the factorial of a number.

Competitive questions on Structures in Hindi

1. `#include <stdio.h>`

```
2. int fact (int);
3. int main()
4. {
5.     int n,f;
6.     printf("Enter the number whose factorial you want to calculate?");
7.     scanf("%d",&n);
8.     f = fact(n);
9.     printf("factorial = %d",f);
10.}
11.int fact(int n)
12.{
13.    if (n==0)
14.    {
15.        return 0;
16.    }
17.    else if ( n == 1)
18.    {
19.        return 1;
20.    }
21.    else
22.    {
23.        return n*fact(n-1);
24.    }
25.}
```

Output

Enter the number whose factorial you want to calculate?5
factorial = 120



We can understand the above program of the recursive method call by the figure given below:

```
return 5 * factorial(4) = 120
└─ return 4 * factorial(3) = 24
    └─ return 3 * factorial(2) = 6
        └─ return 2 * factorial(1) = 2
            └─ return 1 * factorial(0) = 1
```

javaTpoint.com

$1 * 2 * 3 * 4 * 5 = 120$

Fig: Recursion

Recursive Function

A recursive function performs the tasks by dividing it into the subtasks. There is a termination condition defined in the function which is satisfied by some specific subtask. After this, the recursion stops and the final result is returned from the function.

The case at which the function doesn't recur is called the base case whereas the instances where the function keeps calling itself to perform a subtask, is called the recursive case. All the recursive functions can be written using this format.

Pseudocode for writing any recursive function is given below.

1. **if** (test_for_base)
2. {
3. **return** some_value;
4. }
5. **else if** (test_for_another_base)
6. {


```
7.   return some_another_value;
8. }
9. else
10.{
11.  // Statements;
12.  recursive call;
13.}
```

Example of recursion in C

Let's see an example to find the nth term of the Fibonacci series.

```
1. #include<stdio.h>
2. int fibonacci(int);
3. void main ()
4. {
5.   int n,f;
6.   printf("Enter the value of n?");
7.   scanf("%d",&n);
8.   f = fibonacci(n);
9.   printf("%d",f);
10.}
11.int fibonacci (int n)
12.{
13.  if (n==0)
14.  {
15.    return 0;
16.  }
17.  else if (n == 1)
18.  {
```

```
19.     return 1;
20. }
21. else
22. {
23.     return fibonacci(n-1)+fibonacci(n-2);
24. }
25.}
```

Output

Enter the value of n?12

144

Memory allocation of Recursive method

Each recursive call creates a new copy of that method in the memory. Once some data is returned by the method, the copy is removed from the memory. Since all the variables and other stuff declared inside function get stored in the stack, therefore a separate stack is maintained at each recursive call. Once the value is returned from the corresponding function, the stack gets destroyed. Recursion involves so much complexity in resolving and tracking the values at each recursive call. Therefore we need to maintain the stack and track the values of the variables defined in the stack.

Let us consider the following example to understand the memory allocation of the recursive functions.

```
1. int display (int n)
2. {
3.     if(n == 0)
4.         return 0; // terminating condition
5.     else
6.     {
7.         printf("%d",n);
8.         return display(n-1); // recursive call
```

9. }

10.}

Storage Classes in C

Storage classes in C are used to determine the lifetime, visibility, memory location, and initial value of a variable. There are four types of storage classes in C

- Automatic
- External
- Static
- Register

Storage Classes	Storage Place	Default Value	Scope	Lifetime
auto	RAM	Garbage Value	Local	Within function
extern	RAM	Zero	Global	Till the end of the main program Maybe declared anywhere in the program
static	RAM	Zero	Local	Till the end of the main program, Retains value between multiple functions call
register	Register	Garbage Value	Local	Within the function

Automatic

- Automatic variables are allocated memory automatically at runtime.
- The visibility of the automatic variables is limited to the block in which they are defined.
- The scope of the automatic variables is limited to the block in which they are defined. The automatic variables are initialized to garbage by default.
- The memory assigned to automatic variables gets freed upon exiting from the block.
- The keyword used for defining automatic variables is `auto`.
- Every local variable is automatic in C by default.

Example 1

```
1. #include <stdio.h>
2. int main()
3. {
4.     int a; //auto
5.     char b;
6.     float c;
7.     printf("%d %c %f",a,b,c); // printing initial default value of automatic variables a, b,
    and c.
8.     return 0;
9. }
```

Output:

garbage garbage garbage

Example 2

```
1. #include <stdio.h>
2. int main()
3. {
4.     int a = 10,i;
5.     printf("%d ",++a);
6.     {
7.         int a = 20;
8.         for (i=0;i<3;i++)
9.         {
10.            printf("%d ",a); // 20 will be printed 3 times since it is the local value of a
11.        }
12.    }
13.    printf("%d ",a); // 11 will be printed since the scope of a = 20 is ended.
14.}
```

Output:

11 20 20 20 11

Static

- The variables defined as static specifier can hold their value between the multiple function calls.
- Static local variables are visible only to the function or the block in which they are defined.

- A same static variable can be declared many times but can be assigned at only one time.
- Default initial value of the static integral variable is 0 otherwise null.
- The visibility of the static global variable is limited to the file in which it has declared.
- The keyword used to define static variable is static.

Example 1

```

1. #include<stdio.h>
2. static char c;
3. static int i;
4. static float f;
5. static char s[100];
6. void main ()
7. {
8. printf("%d %d %f %s",c,i,f); // the initial default value of c, i, and f will be printed.
9. }
```

Output:

0 0 0.000000 (null)

Example 2

```

1. #include<stdio.h>
2. void sum()
3. {
4. static int a = 10;
5. static int b = 24;
```

```

6. printf("%d %d \n",a,b);
7. a++;
8. b++;
9. }
10. void main()
11. {
12. int i;
13. for(i = 0; i < 3; i++)
14. {
15. sum(); // The static variables holds their value between multiple function calls.
16. }
17. }

```

Output:

```

10 24
11 25
12 26

```

Register

- The variables defined as the register is allocated the memory into the CPU registers depending upon the size of the memory remaining in the CPU.
- We can not dereference the register variables, i.e., we can not use &operator for the register variable.
- The access time of the register variables is faster than the automatic variables.
- The initial default value of the register local variables is 0.

- The register keyword is used for the variable which should be stored in the CPU register. However, it is compiler's choice whether or not; the variables can be stored in the register.
- We can store pointers into the register, i.e., a register can store the address of a variable.
- Static variables can not be stored into the register since we can not use more than one storage specifier for the same variable.

Example 1

1. `#include <stdio.h>`
2. `int main()`
3. `{`
4. `register int a; // variable a is allocated memory in the CPU register. The initial default value of a is 0.`
5. `printf("%d",a);`
6. `}`

Output:

0

Example 2

1. `#include <stdio.h>`
2. `int main()`
3. `{`
4. `register int a = 0;`
5. `printf("%u",&a); // This will give a compile time error since we can not access the address of a register variable.`
6. `}`

Output:

```
main.c:5:5: error: address of register variable 'a' requested
printf("%u",&a);
~~~~~
```



External

- The external storage class is used to tell the compiler that the variable defined as extern is declared with an external linkage elsewhere in the program.
- The variables declared as extern are not allocated any memory. It is only declaration and intended to specify that the variable is declared elsewhere in the program.
- The default initial value of external integral type is 0 otherwise null.
- We can only initialize the extern variable globally, i.e., we can not initialize the external variable within any block or method.
- An external variable can be declared many times but can be initialized at only once.
- If a variable is declared as external then the compiler searches for that variable to be initialized somewhere in the program which may be extern or static. If it is not, then the compiler will show an error.

Example 1

1. `#include <stdio.h>`
2. `int main()`
3. `{`
4. `extern int a;`
5. `printf("%d",a);`
6. `}`

Output

```
main.c:(.text+0x6): undefined reference to `a'
collect2: error: ld returned 1 exit status
```

Example 2

1. `#include <stdio.h>`
2. `int a;`
3. `int main()`
4. `{`
5. `extern int a; // variable a is defined globally, the memory will not be allocated to a`
6. `printf("%d",a);`
7. `}`

Output

```
0
```

Example 3

1. `#include <stdio.h>`
2. `int a;`
3. `int main()`
4. `{`
5. `extern int a = 0; // this will show a compiler error since we can not use extern and initializer at same time`
6. `printf("%d",a);`
7. `}`

Output

```
compile time error
```

```
main.c: In function ?main?:
main.c:5:16: error: ?a? has both ?extern? and initializer
extern int a = 0;
```

Example 4

1. `#include <stdio.h>`
2. `int main()`
3. `{`
4. `extern int a; // Compiler will search here for a variable a defined and initialized somewhere in the program or not.`
5. `printf("%d",a);`
6. `}`
7. `int a = 20;`

Output

```
20
```

Example 5

1. `extern int a;`
2. `int a = 10;`
3. `#include <stdio.h>`
4. `int main()`
5. `{`
6. `printf("%d",a);`
7. `}`
8. `int a = 20; // compiler will show an error at this line`

C Array

An array is defined as the collection of similar type of data items stored at contiguous memory locations. Arrays are the derived data type in C programming language which can store the primitive type of data such as int, char, double, float, etc. It also has the capability to store the collection of derived data types, such as pointers, structure, etc. The array is the simplest data structure where each data element can be randomly accessed by using its index number.

C array is beneficial if you have to store similar elements. For example, if we want to store the marks of a student in 6 subjects, then we don't need to define different variables for the marks in the different subject. Instead of that, we can define an array which can store the marks in each subject at the contiguous memory locations.

By using the array, we can access the elements easily. Only a few lines of code are required to access the elements of the array.

Properties of Array

The array contains the following properties.

- Each element of an array is of same data type and carries the same size, i.e., int = 4 bytes.
- Elements of the array are stored at contiguous memory locations where the first element is stored at the smallest memory location.
- Elements of the array can be randomly accessed since we can calculate the address of each element of the array with the given base address and the size of the data element.

Advantage of C Array

1) Code Optimization: Less code to access the data.

2) Ease of traversing: By using the for loop, we can retrieve the elements of an array easily.

3) Ease of sorting: To sort the elements of the array, we need a few lines of code only.

4) Random Access: We can access any element randomly using the array.

Disadvantage of C Array

1) Fixed Size: Whatever size, we define at the time of declaration of the array, we can't exceed the limit. So, it doesn't grow the size dynamically like LinkedList which we will learn later.

Declaration of C Array

We can declare an array in the c language in the following way.

1. `data_type array_name[array_size];`

Now, let us see the example to declare the array.

1. `int marks[5];`

Here, `int` is the *data_type*, `marks` are the *array_name*, and `5` is the *array_size*.

Initialization of C Array

The simplest way to initialize an array is by using the index of each element. We can initialize each element of the array by using the index. Consider the following example.

1. `marks[0]=80; //initialization of array`
2. `marks[1]=60;`
3. `marks[2]=70;`
4. `marks[3]=85;`
5. `marks[4]=75;`

80	60	70	85	75
----	----	----	----	----

marks[0] marks[1] marks[2] marks[3] marks[4]

Initialization of Array

C array example

```
1. #include<stdio.h>
2. int main(){
3. int i=0;
4. int marks[5];//declaration of array
5. marks[0]=80;//initialization of array
6. marks[1]=60;
7. marks[2]=70;
8. marks[3]=85;
9. marks[4]=75;
10. //traversal of array
11. for(i=0;i<5;i++){
12. printf("%d \n",marks[i]);
13. }//end of for loop
14. return 0;
15. }
```

Output

80
60
70
85
75

C Array: Declaration with Initialization

We can initialize the c array at the time of declaration. Let's see the code.

1. `int marks[5]={20,30,40,50,60};`

In such case, there is **no requirement to define the size**. So it may also be written as the following code.

1. `int marks[]={20,30,40,50,60};`

Let's see the C program to declare and initialize the array in C.

1. `#include<stdio.h>`
2. `int main(){`
3. `int i=0;`
4. `int marks[5]={20,30,40,50,60};`*//declaration and initialization of array*
5. *//traversal of array*
6. `for(i=0;i<5;i++){`
7. `printf("%d \n",marks[i]);`
8. `}`
9. `return 0;`
10. `}`

Output

20
30
40
50
60

C Array Example: Sorting an array

In the following program, we are using bubble sort method to sort the array in ascending order.

```
1. #include<stdio.h>
2. void main ()
3. {
4.     int i, j,temp;
5.     int a[10] = { 10, 9, 7, 101, 23, 44, 12, 78, 34, 23};
6.     for(i = 0; i<10; i++)
7.     {
8.         for(j = i+1; j<10; j++)
9.         {
10.            if(a[j] > a[i])
11.            {
12.                temp = a[i];
13.                a[i] = a[j];
14.                a[j] = temp;
15.            }
16.        }
17.    }
18.    printf("Printing Sorted Element List ...\n");
19.    for(i = 0; i<10; i++)
20.    {
21.        printf("%d\n",a[i]);
22.    }
23.}
```


Program to print the largest and second largest element of the array.

```
1. #include<stdio.h>
2. void main ()
3. {
4.     int arr[100],i,n,largest,sec_largest;
5.     printf("Enter the size of the array?");
6.     scanf("%d",&n);
7.     printf("Enter the elements of the array?");
8.     for(i = 0; i<n; i++)
9.     {
10.         scanf("%d",&arr[i]);
11.     }
12.     largest = arr[0];
13.     sec_largest = arr[1];
14.     for(i=0;i<n;i++)
15.     {
16.         if(arr[i]>largest)
17.         {
18.             sec_largest = largest;
19.             largest = arr[i];
20.         }
21.         else if (arr[i]>sec_largest && arr[i]!=largest)
22.         {
23.             sec_largest=arr[i];
24.         }
25.     }
26.     printf("largest = %d, second largest = %d",largest,sec_largest);
```

27.

28.}

Two Dimensional Array in C

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

Declaration of two dimensional Array in C

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

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

Consider the following example.

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

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

Initialization of 2D Array in C

In the 1D array, we don't need to specify the size of the array if the declaration and initialization are being done simultaneously. However, this will not work with 2D arrays. We will have to define at least the second dimension of the array. The two-dimensional array can be declared and defined in the following way.

1. `int arr[4][3]={{1,2,3},{2,3,4},{3,4,5},{4,5,6}};`

Two-dimensional array example in C

1. `#include<stdio.h>`

2. `int main(){`

```

3. int i=0,j=0;
4. int arr[4][3]={{1,2,3},{2,3,4},{3,4,5},{4,5,6}};
5. //traversing 2D array
6. for(i=0;i<4;i++){
7.     for(j=0;j<3;j++){
8.         printf("arr[%d] [%d] = %d \n",i,j,arr[i][j]);
9.     }//end of j
10. }//end of i
11. return 0;
12.}

```

Output

```

arr[0][0] = 1
arr[0][1] = 2
arr[0][2] = 3
arr[1][0] = 2
arr[1][1] = 3
arr[1][2] = 4
arr[2][0] = 3
arr[2][1] = 4
arr[2][2] = 5
arr[3][0] = 4
arr[3][1] = 5
arr[3][2] = 6

```

C 2D array example: Storing elements in a matrix and printing it.

```

1. #include <stdio.h>
2. void main ()
3. {
4.     int arr[3][3],i,j;
5.     for (i=0;i<3;i++)

```

```
6.  {
7.    for (j=0;j<3;j++)
8.    {
9.        printf("Enter a[%d][%d]: ",i,j);
10.        scanf("%d",&arr[i][j]);
11.    }
12. }
13. printf("\n printing the elements ....\n");
14. for(i=0;i<3;i++)
15. {
16.     printf("\n");
17.     for (j=0;j<3;j++)
18.     {
19.         printf("%d\t",arr[i][j]);
20.     }
21. }
22. }
```

Output

```
Enter a[0][0]: 56
Enter a[0][1]: 10
Enter a[0][2]: 30
Enter a[1][0]: 34
Enter a[1][1]: 21
Enter a[1][2]: 34
```

```
Enter a[2][0]: 45
Enter a[2][1]: 56
Enter a[2][2]: 78
```

```
printing the elements ....
```

```
56  10  30
34  21  34
45  56  78
```

Return an Array in C

What is an Array?

An array is a type of data structure that stores a fixed-size of a homogeneous collection of data. In short, we can say that array is a collection of variables of the same type.

For example, if we want to declare 'n' number of variables, n1, n2...n., if we create all these variables individually, then it becomes a very tedious task. In such a case, we create an array of variables having the same type. Each element of an array can be accessed using an index of the element.

Let's first see how to pass a single-dimensional array to a function.

Passing array to a function

```
1. #include <stdio.h>
2. void getarray(int arr[])
3. {
4.     printf("Elements of array are : ");
5.     for(int i=0;i<5;i++)
6.     {
7.         printf("%d ", arr[i]);
8.     }
```

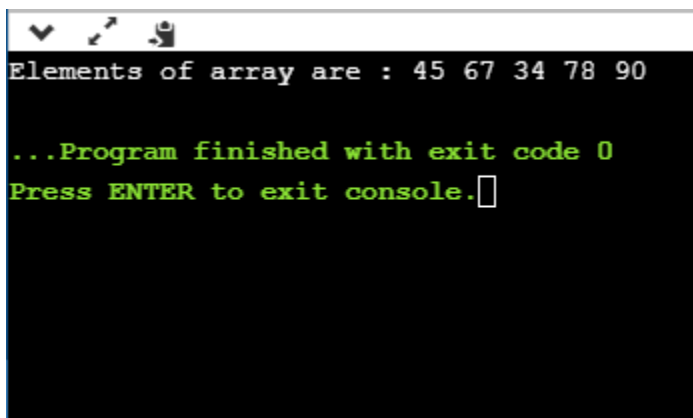
```

9. }
10. int main()
11. {
12.     int arr[5]={45,67,34,78,90};
13.     getarray(arr);
14.     return 0;
15. }

```

In the above program, we have first created the array **arr** and then we pass this array to the function **getarray()**. The **getarray()** function prints all the elements of the array **arr**.

Output



```

Elements of array are : 45 67 34 78 90

...Program finished with exit code 0
Press ENTER to exit console.

```

Passing array to a function as a pointer

Now, we will see how to pass an array to a function as a pointer.

```

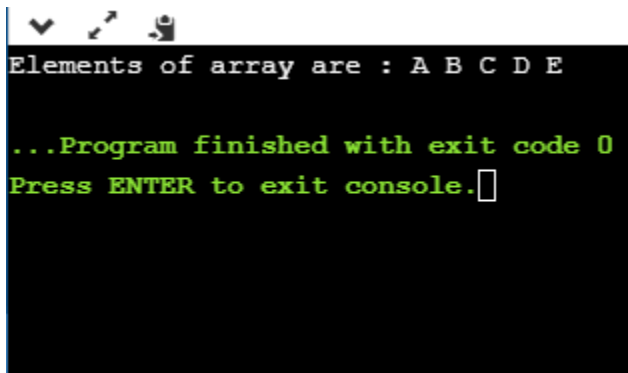
1. #include <stdio.h>
2. void printarray(char *arr)
3. {
4.     printf("Elements of array are : ");
5.     for(int i=0;i<5;i++)
6.     {
7.         printf("%C ", arr[i]);
8.     }

```

```
9. }
10. int main()
11. {
12.     char arr[5]={'A','B','C','D','E'};
13.     printarray(arr);
14.     return 0;
15. }
```

In the above code, we have passed the array to the function as a pointer. The function **printarray()** prints the elements of an array.

Output



```
Elements of array are : A B C D E
...Program finished with exit code 0
Press ENTER to exit console.
```

Note: From the above examples, we observe that array is passed to a function as a reference which means that array also persists outside the function.

How to return an array from a function

Returning pointer pointing to the array

```
1. #include <stdio.h>
2. int *getarray()
3. {
4.     int arr[5];
5.     printf("Enter the elements in an array : ");
```

```
6.  for(int i=0;i<5;i++)
7.  {
8.      scanf("%d", &arr[i]);
9.  }
10. return arr;
11.}
12.int main()
13.{
14. int *n;
15. n=getarray();
16. printf("\nElements of array are :");
17. for(int i=0;i<5;i++)
18. {
19.     printf("%d", n[i]);
20. }
21. return 0;
22.}
```

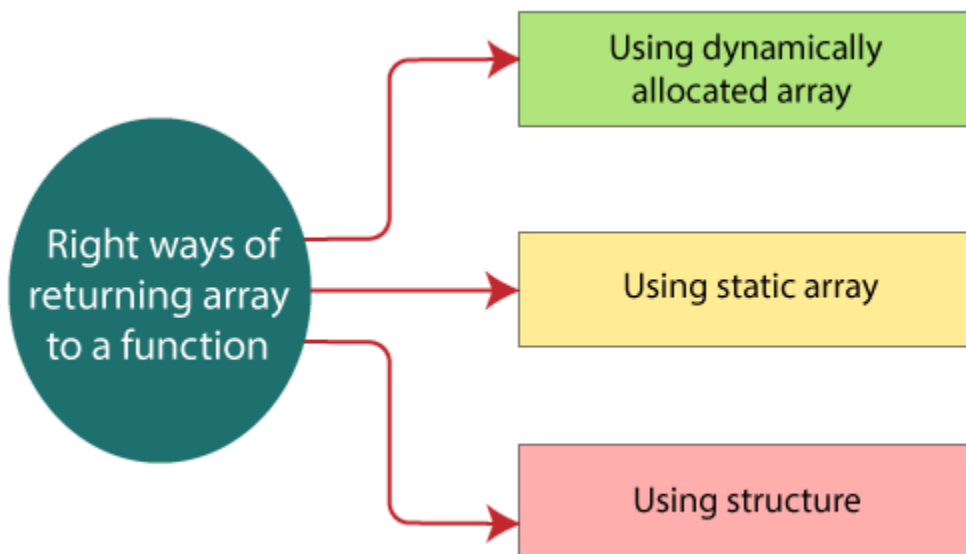
In the above program, **getarray()** function returns a variable 'arr'. It returns a local variable, but it is an illegal memory location to be returned, which is allocated within a function in the stack. Since the program control comes back to the **main()** function, and all the variables in a stack are freed. Therefore, we can say that this program is returning memory location, which is already de-allocated, so the output of the program is a **segmentation fault**.

Output


```
input
main.c:27:12: warning: function returns address of local variable [-Wreturn-local-addr]
Array inside function: 1
2
3
4
5
Array outside function:
Segmentation fault (core dumped)
```

There are three right ways of returning an array to a function:

- Using dynamically allocated array
- Using static array
- Using structure



Returning array by passing an array which is to be returned as a parameter to the function.

1. `#include <stdio.h>`
2. `int *getarray(int *a)`
3. `{`

```
4.
5.  printf("Enter the elements in an array : ");
6.  for(int i=0;i<5;i++)
7.  {
8.      scanf("%d", &a[i]);
9.  }
10. return a;
11.}
12.int main()
13.{
14. int *n;
15. int a[5];
16. n=getarray(a);
17. printf("\nElements of array are :");
18. for(int i=0;i<5;i++)
19. {
20.     printf("%d", n[i]);
21. }
22. return 0;
23.}
```

Output

```
Enter the elements in an array :
1
2
3
4
5

Elements of array are :1 2 3 4 5

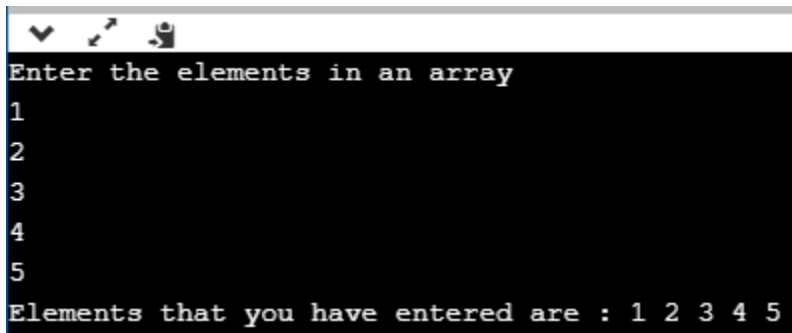
...Program finished with exit code 0
Press ENTER to exit console.
```

Returning array using malloc() function.

1. `#include <stdio.h>`
2. `#include <malloc.h>`
3. `int *getarray()`
4. `{`
5. `int size;`
6. `printf("Enter the size of the array : ");`
7. `scanf("%d",&size);`
8. `int *p= malloc(sizeof(size));`
9. `printf("\nEnter the elements in an array");`
10. `for(int i=0;i<size;i++)`
11. `{`
12. `scanf("%d",&p[i]);`
13. `}`
14. `return p;`
15. `}`
16. `int main()`
17. `{`
18. `int *ptr;`
19. `ptr=getarray();`

```
20. int length=sizeof(*ptr);
21. printf("Elements that you have entered are : ");
22. for(int i=0;ptr[i]!='\0';i++)
23. {
24.     printf("%d ", ptr[i]);
25. }
26. return 0;
27.}
```

Output



```
Enter the elements in an array
1
2
3
4
5
Elements that you have entered are : 1 2 3 4 5
```

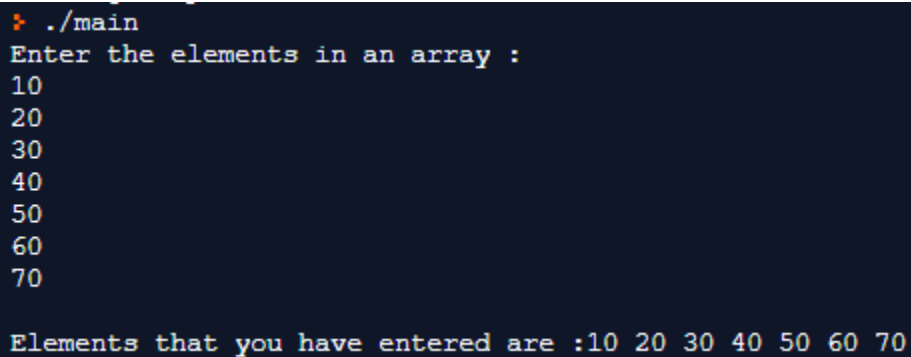
Using Static Variable

```
1. #include <stdio.h>
2. int *getarray()
3. {
4.     static int arr[7];
5.     printf("Enter the elements in an array : ");
6.     for(int i=0;i<7;i++)
7.     {
8.         scanf("%d",&arr[i]);
9.     }
```

```
10. return arr;
11.
12.}
13.int main()
14.{
15. int *ptr;
16. ptr=getarray();
17. printf("\nElements that you have entered are :");
18. for(int i=0;i<7;i++)
19. {
20.     printf("%d ", ptr[i]);
21. }
22.}
```

In the above code, we have created the variable **arr[]** as static in **getarray()** function, which is available throughout the program. Therefore, the function **getarray()** returns the actual memory location of the variable '**arr**'.

Output



```
➤ ./main
Enter the elements in an array :
10
20
30
40
50
60
70

Elements that you have entered are :10 20 30 40 50 60 70
```

Using Structure

The structure is a user-defined data type that can contain a collection of items of different types. Now, we will create a program that returns an array by using structure.

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

```
2. #include<malloc.h>
3. struct array
4. {
5.     int arr[8];
6. };
7. struct array getarray()
8. {
9.     struct array y;
10.    printf("Enter the elements in an array : ");
11.    for(int i=0;i<8;i++)
12.    {
13.        scanf("%d",&y.arr[i]);
14.    }
15.    return y;
16.}
17.int main()
18.{
19.    struct array x=getarray();
20.    printf("Elements that you have entered are :");
21.    for(int i=0;x.arr[i]!='\0';i++)
22.    {
23.        printf("%d ", x.arr[i]);
24.    }
25.    return 0;
26.}
```

Output

```
Enter the elements in an array :
4
5
6
7
8
9
1
2
Elements that you have entered are :4 5 6 7 8 9 1 2
```

Passing Array to Function in C

In C, there are various general problems which requires passing more than one variable of the same type to a function. For example, consider a function which sorts the 10 elements in ascending order. Such a function requires 10 numbers to be passed as the actual parameters from the main function. Here, instead of declaring 10 different numbers and then passing into the function, we can declare and initialize an array and pass that into the function. This will resolve all the complexity since the function will now work for any number of values.

As we know that the array_name contains the address of the first element. Here, we must notice that we need to pass only the name of the array in the function which is intended to accept an array. The array defined as the formal parameter will automatically refer to the array specified by the array name defined as an actual parameter.

Consider the following syntax to pass an array to the function.

1. `functionname(arrayname);` **//passing array**

Methods to declare a function that receives an array as an argument

There are 3 ways to declare the function which is intended to receive an array as an argument.

First way:

1. `return_type function(type arrayname[])`

Declaring blank subscript notation `[]` is the widely used technique.

Second way:

1. `return_type function(type arrayname[SIZE])`

Optionally, we can define size in subscript notation `[]`.

Third way:

1. `return_type function(type *arrayname)`

You can also use the concept of a pointer. In pointer chapter, we will learn about it.

C language passing an array to function example

1. `#include<stdio.h>`
2. `int minarray(int arr[],int size){`
3. `int min=arr[0];`
4. `int i=0;`
5. `for(i=1;i<size;i++){`
6. `if(min>arr[i]){`
7. `min=arr[i];`
8. `}`
9. `}//end of for`
10. `return min;`
11. `}//end of function`
- 12.
13. `int main(){`


```

14.int i=0,min=0;
15.int numbers[]={4,5,7,3,8,9};//declaration of array
16.
17.min=minarray(numbers,6);//passing array with size
18.printf("minimum number is %d \n",min);
19.return 0;
20.}

```

Output

minimum number is 3

C function to sort the array

```

1. #include<stdio.h>
2. void Bubble_Sort(int[]);
3. void main ()
4. {
5.     int arr[10] = { 10, 9, 7, 101, 23, 44, 12, 78, 34, 23};
6.     Bubble_Sort(arr);
7. }
8. void Bubble_Sort(int a[]) //array a[] points to arr.
9. {
10.int i, j,temp;
11. for(i = 0; i<10; i++)
12. {
13.     for(j = i+1; j<10; j++)
14.     {
15.         if(a[j] < a[i])
16.         {

```

```
17.      temp = a[i];
18.      a[i] = a[j];
19.      a[j] = temp;
20.  }
21.  }
22. }
23. printf("Printing Sorted Element List ...\n");
24. for(i = 0; i<10; i++)
25. {
26.     printf("%d\n",a[i]);
27. }
28.}
```

Output



```
Printing Sorted Element List ...
7
9
10
12
23
23
34
44
78
101
```

Returning array from the function

As we know that, a function can not return more than one value. However, if we try to write the return statement as `return a, b, c;` to return three values (a,b,c), the function will return the last mentioned value which is c in our case. In some problems, we may need to return multiple values from a function. In such cases, an array is returned from the function.

Returning an array is similar to passing the array into the function. The name of the array is returned from the function. To make a function returning an array, the following syntax is used.

1. `int * Function_name() {`
2. `//some statements;`
3. `return array_type;`
4. `}`

To store the array returned from the function, we can define a pointer which points to that array. We can traverse the array by increasing that pointer since pointer initially points to the base address of the array. Consider the following example that contains a function returning the sorted array.

1. `#include<stdio.h>`
2. `int* Bubble_Sort(int[]);`
3. `void main ()`
4. `{`
5. `int arr[10] = { 10, 9, 7, 101, 23, 44, 12, 78, 34, 23};`
6. `int *p = Bubble_Sort(arr), i;`
7. `printf("printing sorted elements ...\n");`
8. `for(i=0;i<10;i++)`
9. `{`
10. `printf("%d\n",*(p+i));`
11. `}`
12. `}`
13. `int* Bubble_Sort(int a[]) //array a[] points to arr.`
14. `{`

```

15.int i, j,temp;
16.  for(i = 0; i<10; i++)
17.  {
18.      for(j = i+1; j<10; j++)
19.      {
20.          if(a[j] < a[i])
21.          {
22.              temp = a[i];
23.              a[i] = a[j];
24.              a[j] = temp;
25.          }
26.      }
27.  }
28.  return a;
29.}

```

Output

Printing Sorted Element List ...

7
9
10
12
23
23
34
44
78

101

Matrix of array multiplication

```

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

```

```

int a[10][10],b[10][10],mul[10][10],r,c,i,j,k;
system("cls");
printf("enter the number of row=");
scanf("%d",&r);
printf("enter the number of column=");
scanf("%d",&c);
printf("enter the first matrix element=\n");
for(i=0;i<r;i++)
{
for(j=0;j<c;j++)
{
scanf("%d",&a[i][j]);
}
}
printf("enter the second matrix element=\n");
for(i=0;i<r;i++)
{
for(j=0;j<c;j++)
{
scanf("%d",&b[i][j]);
}
}

printf("multiply of the matrix=\n");
for(i=0;i<r;i++)
{
for(j=0;j<c;j++)
{
mul[i][j]=0;
for(k=0;k<c;k++)
{
mul[i][j]+=a[i][k]*b[k][j];
}
}
}
//for printing result
for(i=0;i<r;i++)
{
for(j=0;j<c;j++)

```

```

{
printf("%d\t",mul[i][j]);
}
printf("\n");
}
return 0;
}

```

Addition

1. `#include <stdio.h>`
- 2.
3. `int main()`
4. `{`
5. `int rows, cols;`
- 6.
7. `//Initialize matrix a`
8. `int a[][3] = {`
9. `{1, 0, 1},`
10. `{4, 5, 6},`
11. `{1, 2, 3}`
12. `};`
- 13.
14. `//Initialize matrix b`
15. `int b[][3] = {`
16. `{1, 1, 1},`
17. `{2, 3, 1},`
18. `{1, 5, 1}`
19. `};`
- 20.
21. `//Calculates number of rows and columns present in given matrix`

```

22. rows = (sizeof(a)/sizeof(a[0]));
23. cols = (sizeof(a)/sizeof(a[0][0]))/rows;
24.
25. //Array sum will hold the result
26. int sum[rows][cols];
27.
28. //Performs addition of matrices a and b. Store the result in matrix sum
29. for(int i = 0; i < rows; i++){
30.     for(int j = 0; j < cols; j++){
31.         sum[i][j] = a[i][j] + b[i][j];
32.     }
33. }
34.
35. printf("Addition of two matrices: \n");
36. for(int i = 0; i < rows; i++){
37.     for(int j = 0; j < cols; j++){
38.         printf("%d ", sum[i][j]);
39.     }
40.     printf("\n");
41. }
42.
43. return 0;
44.}

```

C Pointers

The pointer in C language is a variable which stores the address of another variable. This variable can be of type int, char, array, function, or any other pointer. The size of

the pointer depends on the architecture. However, in 32-bit architecture the size of a pointer is 2 byte.

Consider the following example to define a pointer which stores the address of an integer.

1. `int n = 10;`
2. `int* p = &n; // Variable p of type pointer is pointing to the address of the variable n of type integer.`

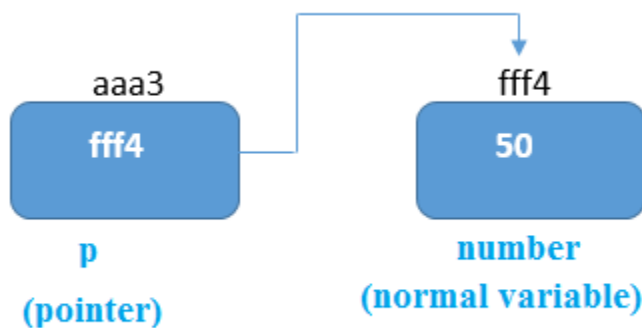
Declaring a pointer

The pointer in c language can be declared using * (asterisk symbol). It is also known as indirection pointer used to dereference a pointer.

1. `int *a; //pointer to int`
2. `char *c; //pointer to char`

Pointer Example

An example of using pointers to print the address and value is given below.



javatpoint.com

As you can see in the above figure, pointer variable stores the address of number variable, i.e., fff4. The value of number variable is 50. But the address of pointer variable p is aaa3.

By the help of * (indirection operator), we can print the value of pointer variable p.

Let's see the pointer example as explained for the above figure.

1. `#include<stdio.h>`
2. `int main(){`
3. `int number=50;`
4. `int *p;`
5. `p=&number;` // stores the address of number variable
6. `printf("Address of p variable is %x \n",p);` // p contains the address of the number therefore printing p gives the address of number.
7. `printf("Value of p variable is %d \n",*p);` // As we know that * is used to dereference a pointer therefore if we print *p, we will get the value stored at the address contained by p.
8. `return 0;`
9. `}`

Output

```
Address of number variable is fff4
Address of p variable is fff4
Value of p variable is 50
```

Pointer to array

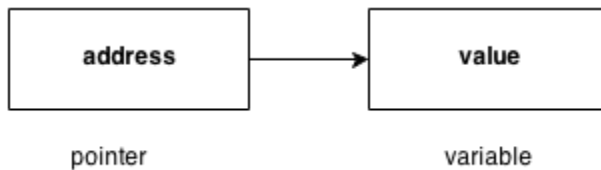
1. `int arr[10];`
2. `int *p[10]=&arr;` // Variable p of type pointer is pointing to the address of an integer array arr.

Pointer to a function

1. `void show (int);`
2. `void(*p)(int) = &display;` // Pointer p is pointing to the address of a function

Pointer to structure

1. `struct st {`
2. `int i;`
3. `float f;`
4. `}ref;`
5. `struct st *p = &ref;`



Advantage of pointer

- 1) Pointer reduces the code and improves the performance, it is used to retrieving strings, trees, etc. and used with arrays, structures, and functions.
- 2) We can return multiple values from a function using the pointer.
- 3) It makes you able to access any memory location in the computer's memory.

Usage of pointer

There are many applications of pointers in c language.

1) Dynamic memory allocation

In c language, we can dynamically allocate memory using `malloc()` and `calloc()` functions where the pointer is used.

2) Arrays, Functions, and Structures

Pointers in c language are widely used in arrays, functions, and structures. It reduces the code and improves the performance.

Address Of (&) Operator

The address of operator '&' returns the address of a variable. But, we need to use %u to display the address of a variable.

```
1. #include<stdio.h>
2. int main(){
3. int number=50;
4. printf("value of number is %d, address of number is %u",number,&number);
5. return 0;
6. }
```

Output

value of number is 50, address of number is fff4

NULL Pointer

A pointer that is not assigned any value but NULL is known as the NULL pointer. If you don't have any address to be specified in the pointer at the time of declaration, you can assign NULL value. It will provide a better approach.

```
int *p=NULL;
```

In the most libraries, the value of the pointer is 0 (zero).

Pointer Program to swap two numbers without using the 3rd variable.

```
1. #include<stdio.h>
2. int main(){
3. int a=10,b=20,*p1=&a,*p2=&b;
4.
5. printf("Before swap: *p1=%d *p2=%d",*p1,*p2);
```

```

6. *p1=*p1+*p2;//30
7. *p2=*p1-*p2; //10
8. *p1=*p1-*p2; //20
9. printf("\nAfter swap: *p1=%d *p2=%d",*p1,*p2);
10.
11.return 0;
12.}

```

Output

Before swap: *p1=10 *p2=20
 After swap: *p1=20 *p2=10

Reading complex pointers

There are several things which must be taken into the consideration while reading the complex pointers in C. Lets see the precedence and associativity of the operators which are used regarding pointers.

Operator	Precedence	Associativity
() , []	1	Left to right
*, identifier	2	Right to left
Data type	3	-

Here,we must notice that,

- **()**: This operator is a bracket operator used to declare and define the function.
- **[]**: This operator is an array subscript operator
- *****: This operator is a pointer operator.

- **Identifier:** It is the name of the pointer. The priority will always be assigned to this.
- **Data type:** Data type is the type of the variable to which the pointer is intended to point. It also includes the modifier like signed int, long, etc).

How to read the pointer: `int (*p)[10]`.

To read the pointer, we must see that `()` and `[]` have the equal precedence. Therefore, their associativity must be considered here. The associativity is left to right, so the priority goes to `()`.

Inside the bracket `()`, pointer operator `*` and pointer name (identifier) `p` have the same precedence. Therefore, their associativity must be considered here which is right to left, so the priority goes to `p`, and the second priority goes to `*`.

Assign the 3rd priority to `[]` since the data type has the last precedence. Therefore the pointer will look like following.

- `char` -> 4
- `*` -> 2
- `p` -> 1
- `[10]` -> 3

The pointer will be read as `p` is a pointer to an array of integers of size 10.

Example

How to read the following pointer?

1. `int (*p)(int (*)[2], int (*)void)`

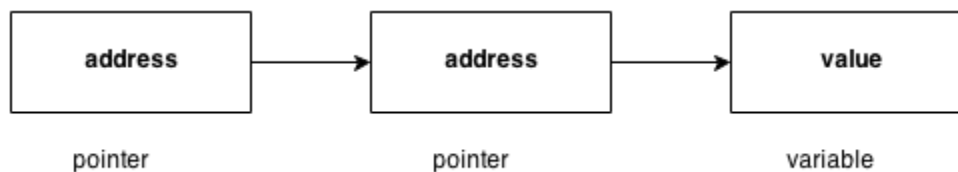
Explanation

This pointer will be read as `p` is a pointer to such function which accepts the first parameter as the pointer to a one-dimensional array of integers of size two and the

second parameter as the pointer to a function which parameter is void and return type is the integer.

C Double Pointer (Pointer to Pointer)

As we know that, a pointer is used to store the address of a variable in C. Pointer reduces the access time of a variable. However, In C, we can also define a pointer to store the address of another pointer. Such pointer is known as a double pointer (pointer to pointer). The first pointer is used to store the address of a variable whereas the second pointer is used to store the address of the first pointer. Let's understand it by the diagram given below.



The syntax of declaring a double pointer is given below.

1. `int **p; // pointer to a pointer which is pointing to an integer.`

Consider the following example.

1. `#include<stdio.h>`
2. `void main ()`
3. `{`
4. `int a = 10;`
5. `int *p;`
6. `int **pp;`
7. `p = &a; // pointer p is pointing to the address of a`
8. `pp = &p; // pointer pp is a double pointer pointing to the address of pointer p`
9. `printf("address of a: %x\n",p); // Address of a will be printed`
10. `printf("address of p: %x\n",pp); // Address of p will be printed`
11. `printf("value stored at p: %d\n",*p); // value stored at the address contained by p i.e. 10 will be printed`

```

12. printf("value stored at pp: %d\n",**pp); // value stored at the address
    contained by the pointer stoyred at pp
13.}

```

Output

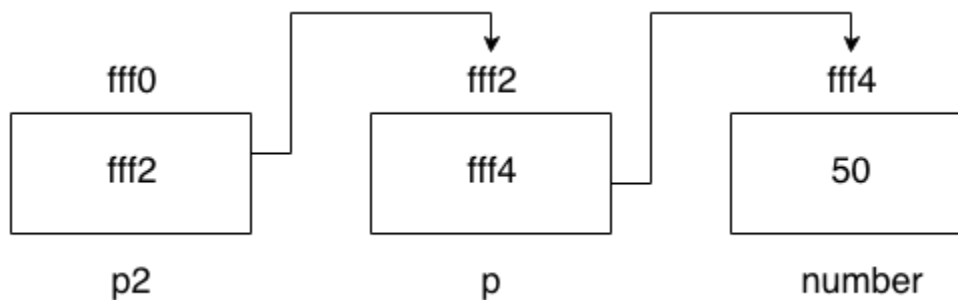
```

address of a: d26a8734
address of p: d26a8738
value stored at p: 10
value stored at pp: 10

```

C double pointer example

Let's see an example where one pointer points to the address of another pointer.



As you can see in the above figure, p2 contains the address of p (fff2), and p contains the address of number variable (fff4).

1. `#include<stdio.h>`
2. `int main(){`
3. `int number=50;`
4. `int *p;//pointer to int`
5. `int **p2;//pointer to pointer`
6. `p=&number;//stores the address of number variable`
7. `p2=&p;`
8. `printf("Address of number variable is %x \n",&number);`

```

9. printf("Address of p variable is %x \n",p);
10.printf("Value of *p variable is %d \n",*p);
11.printf("Address of p2 variable is %x \n",p2);
12.printf("Value of **p2 variable is %d \n",*p);
13.return 0;
14.}

```

Output

```

Address of number variable is fff4
Address of p variable is fff4
Value of *p variable is 50
Address of p2 variable is fff2
Value of **p variable is 50

```

Q. What will be the output of the following program?

```

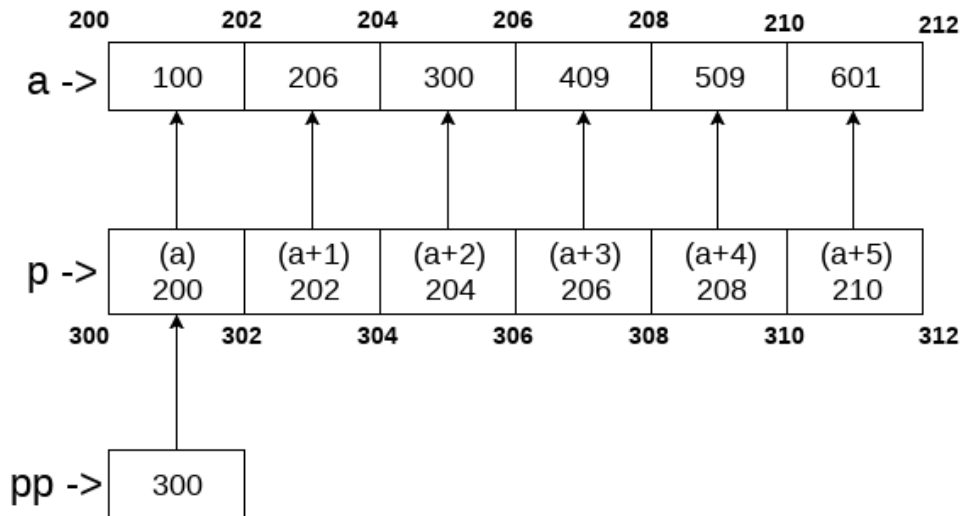
1. #include<stdio.h>
2. void main ()
3. {
4.     int a[10] = {100, 206, 300, 409, 509, 601}; //Line 1
5.     int *p[] = {a, a+1, a+2, a+3, a+4, a+5}; //Line 2
6.     int **pp = p; //Line 3
7.     pp++; // Line 4
8.     printf("%d %d %d\n",pp-p,*pp - a,**pp); // Line 5
9.     *pp++; // Line 6
10.    printf("%d %d %d\n",pp-p,*pp - a,**pp); // Line 7
11.    ++*pp; // Line 8
12.    printf("%d %d %d\n",pp-p,*pp - a,**pp); // Line 9
13.    ++**pp; // Line 10

```



```
14. printf("%d %d %d\n",pp-p,*pp - a,**pp); // Line 11
15.}
```

Explanation



To access $a[0]$ $\longrightarrow a[0] = *(a) = *p[0] = **(\text{pp}+0) = **(\text{pp}+0) = 100$

In the above question, the pointer arithmetic is used with the double pointer. An array of 6 elements is defined which is pointed by an array of pointer p . The pointer array p is pointed by a double pointer pp . However, the above image gives you a brief idea about how the memory is being allocated to the array a and the pointer array p . The elements of p are the pointers that are pointing to every element of the array a . Since we know that the array name contains the base address of the array hence, it will work as a pointer and can the value can be traversed by using $*(a)$, $*(a+1)$, etc. As shown in the image, $a[0]$ can be accessed in the following ways.

- $a[0]$: it is the simplest way to access the first element of the array
- $*(a)$: since a store the address of the first element of the array, we can access its value by using indirection pointer on it.
- $*p[0]$: if $a[0]$ is to be accessed by using a pointer p to it, then we can use indirection operator ($*$) on the first element of the pointer array p , i.e., $*p[0]$.

- ****pp**: as pp stores the base address of the pointer array, *pp will give the value of the first element of the pointer array that is the address of the first element of the integer array. **p will give the actual value of the first element of the integer array.

Coming to the program, Line 1 and 2 declare the integer and pointer array relatively. Line 3 initializes the double pointer to the pointer array p. As shown in the image, if the address of the array starts from 200 and the size of the integer is 2, then the pointer array will contain the values as 200, 202, 204, 206, 208, 210. Let us consider that the base address of the pointer array is 300; the double pointer pp contains the address of pointer array, i.e., 300. Line number 4 increases the value of pp by 1, i.e., pp will now point to address 302.

Line number 5 contains an expression which prints three values, i.e., pp - p, *pp - a, **pp. Let's calculate them each one of them.

- $pp = 302, p = 300 \Rightarrow pp - p = (302 - 300) / 2 \Rightarrow pp - p = 1$, i.e., 1 will be printed.
- $pp = 302, *pp = 202, a = 200 \Rightarrow *pp - a = 202 - 200 = 2 / 2 = 1$, i.e., 1 will be printed.
- $pp = 302, *pp = 202, *(*pp) = 206$, i.e., 206 will be printed.

Therefore as the result of line 5, The output 1, 1, 206 will be printed on the console. On line 6, *pp++ is written. Here, we must notice that two unary operators * and ++ will have the same precedence. Therefore, by the rule of associativity, it will be evaluated from right to left. Therefore the expression *pp++ can be rewritten as (*(pp++)). Since, pp = 302 which will now become, 304. *pp will give 204.

On line 7, again the expression is written which prints three values, i.e., pp-p, *pp-a, **pp. Let's calculate each one of them.

- $pp = 304, p = 300 \Rightarrow pp - p = (304 - 300) / 2 \Rightarrow pp - p = 2$, i.e., 2 will be printed.
- $pp = 304, *pp = 204, a = 200 \Rightarrow *pp - a = (204 - 200) / 2 = 2$, i.e., 2 will be printed.
- $pp = 304, *pp = 204, *(*pp) = 300$, i.e., 300 will be printed.

Therefore, as the result of line 7, The output 2, 2, 300 will be printed on the console. On line 8, `++*pp` is written. According to the rule of associativity, this can be rewritten as, `(++(*pp))`. Since, `pp = 304`, `*pp = 204`, the value of `*pp = *(p[2]) = 206` which will now point to `a[3]`.

On line 9, again the expression is written which prints three values, i.e., `pp-p`, `*pp-a`, `*pp`. Let's calculate each one of them.

- `pp = 304, p = 300 => pp - p = (304 - 300)/2 => pp-p = 2`, i.e., 2 will be printed.
- `pp = 304, *pp = 206, a = 200 => *pp-a = (206 - 200)/2 = 3`, i.e., 3 will be printed.
- `pp = 304, *pp = 206, *(*pp) = 409`, i.e., 409 will be printed.

Therefore, as the result of line 9, the output 2, 3, 409 will be printed on the console. On line 10, `++**pp` is written. according to the rule of associativity, this can be rewritten as, `(++(*(*pp)))`. `pp = 304, *pp = 206, **pp = 409, ++**pp => *pp = *pp + 1 = 410`. In other words, `a[3] = 410`.

On line 11, again the expression is written which prints three values, i.e., `pp-p`, `*pp-a`, `*pp`. Let's calculate each one of them.

- `pp = 304, p = 300 => pp - p = (304 - 300)/2 => pp-p = 2`, i.e., 2 will be printed.
- `pp = 304, *pp = 206, a = 200 => *pp-a = (206 - 200)/2 = 3`, i.e., 3 will be printed.
- On line 8, `**pp = 410`.

Therefore as the result of line 9, the output 2, 3, 410 will be printed on the console.

At last, the output of the complete program will be given as:

Output

1 1 206

2 2 300

2 3 409

2 3 410

Pointer Arithmetic in C

We can perform arithmetic operations on the pointers like addition, subtraction, etc. However, as we know that pointer contains the address, the result of an arithmetic operation performed on the pointer will also be a pointer if the other operand is of type integer. In pointer-from-pointer subtraction, the result will be an integer value. Following arithmetic operations are possible on the pointer in C language:

- Increment
 - Decrement
 - Addition
 - Subtraction
 - Comparison
-

Incrementing Pointer in C

If we increment a pointer by 1, the pointer will start pointing to the immediate next location. This is somewhat different from the general arithmetic since the value of the pointer will get increased by the size of the data type to which the pointer is pointing.

We can traverse an array by using the increment operation on a pointer which will keep pointing to every element of the array, perform some operation on that, and update itself in a loop.

The Rule to increment the pointer is given below:

1. $\text{new_address} = \text{current_address} + i * \text{size_of}(\text{data type})$

Where i is the number by which the pointer get increased.

32-bit

For 32-bit int variable, it will be incremented by 2 bytes.

64-bit

For 64-bit int variable, it will be incremented by 4 bytes.

Let's see the example of incrementing pointer variable on 64-bit architecture.

1. `#include<stdio.h>`
2. `int main(){`
3. `int number=50;`
4. `int *p;//pointer to int`
5. `p=&number;//stores the address of number variable`
6. `printf("Address of p variable is %u \n",p);`
7. `p=p+1;`
8. `printf("After increment: Address of p variable is %u \n",p); // in our case, p will get incremented by 4 bytes.`
9. `return 0;`
10. `}`

Output

Address of p variable is 3214864300

After increment: Address of p variable is 3214864304

Traversing an array by using pointer

1. `#include<stdio.h>`
2. `void main ()`
3. `{`
4. `int arr[5] = {1, 2, 3, 4, 5};`
5. `int *p = arr;`
6. `int i;`

```

7.  printf("printing array elements...\n");
8.  for(i = 0; i < 5; i++)
9.  {
10.     printf("%d ",*(p+i));
11. }
12.}

```

Output

```

printing array elements...
1 2 3 4 5

```

Decrementing Pointer in C

Like increment, we can decrement a pointer variable. If we decrement a pointer, it will start pointing to the previous location. The formula of decrementing the pointer is given below:

$$1. \text{ new_address} = \text{current_address} - i * \text{size_of}(\text{data type})$$

32-bit

For 32-bit int variable, it will be decremented by 2 bytes.

64-bit

For 64-bit int variable, it will be decremented by 4 bytes.

Let's see the example of decrementing pointer variable on 64-bit OS.

```

1. #include <stdio.h>
2. void main(){

```

3. `int number=50;`
4. `int *p;//pointer to int`
5. `p=&number;//stores the address of number variable`
6. `printf("Address of p variable is %u \n",p);`
7. `p=p-1;`
8. `printf("After decrement: Address of p variable is %u \n",p); // P will now point to the immediate previous location.`
9. `}`

Output

Address of p variable is 3214864300

After decrement: Address of p variable is 3214864296

C Pointer Addition

We can add a value to the pointer variable. The formula of adding value to pointer is given below:

1. $\text{new_address} = \text{current_address} + (\text{number} * \text{size_of}(\text{data type}))$

32-bit

For 32-bit int variable, it will add $2 * \text{number}$.

64-bit

For 64-bit int variable, it will add $4 * \text{number}$.

Let's see the example of adding value to pointer variable on 64-bit architecture.

1. `#include<stdio.h>`
2. `int main(){`
3. `int number=50;`

```
4. int *p;//pointer to int
5. p=&number;//stores the address of number variable
6. printf("Address of p variable is %u \n",p);
7. p=p+3; //adding 3 to pointer variable
8. printf("After adding 3: Address of p variable is %u \n",p);
9. return 0;
10.}
```

Output

```
Address of p variable is 3214864300
After adding 3: Address of p variable is 3214864312
```

As you can see, the address of p is 3214864300. But after adding 3 with p variable, it is 3214864312, i.e., $4 \times 3 = 12$ increment. Since we are using 64-bit architecture, it increments 12. But if we were using 32-bit architecture, it was incrementing to 6 only, i.e., $2 \times 3 = 6$. As integer value occupies 2-byte memory in 32-bit OS.

C Pointer Subtraction

Like pointer addition, we can subtract a value from the pointer variable. Subtracting any number from a pointer will give an address. The formula of subtracting value from the pointer variable is given below:

1. $\text{new_address} = \text{current_address} - (\text{number} * \text{size_of}(\text{data type}))$

32-bit

For 32-bit int variable, it will subtract $2 * \text{number}$.

64-bit

For 64-bit int variable, it will subtract $4 * \text{number}$.

Let's see the example of subtracting value from the pointer variable on 64-bit architecture.

```
1. #include<stdio.h>
2. int main(){
3. int number=50;
4. int *p;//pointer to int
5. p=&number;//stores the address of number variable
6. printf("Address of p variable is %u \n",p);
7. p=p-3; //subtracting 3 from pointer variable
8. printf("After subtracting 3: Address of p variable is %u \n",p);
9. return 0;
10.}
```

Output

```
Address of p variable is 3214864300
After subtracting 3: Address of p variable is 3214864288
```

You can see after subtracting 3 from the pointer variable, it is 12 (4*3) less than the previous address value.

However, instead of subtracting a number, we can also subtract an address from another address (pointer). This will result in a number. It will not be a simple arithmetic operation, but it will follow the following rule.

If two pointers are of the same type,

1. $\text{Address2} - \text{Address1} = (\text{Subtraction of two addresses}) / \text{size of data type which pointer points}$

Consider the following example to subtract one pointer from another.

```
1. #include<stdio.h>
2. void main ()
```

```
3. {  
4.   int i = 100;  
5.   int *p = &i;  
6.   int *temp;  
7.   temp = p;  
8.   p = p + 3;  
9.   printf("Pointer Subtraction: %d - %d = %d",p, temp, p-temp);  
10.}
```

Output

Pointer Subtraction: 1030585080 - 1030585068 = 3

Illegal arithmetic with pointers

There are various operations which can not be performed on pointers. Since, pointer stores address hence we must ignore the operations which may lead to an illegal address, for example, addition, and multiplication. A list of such operations is given below.

- Address + Address = illegal
- Address * Address = illegal
- Address % Address = illegal
- Address / Address = illegal
- Address & Address = illegal
- Address ^ Address = illegal
- Address | Address = illegal
- ~Address = illegal

Pointer to function in C

As we discussed in the previous chapter, a pointer can point to a function in C. However, the declaration of the pointer variable must be the same as the function. Consider the following example to make a pointer pointing to the function.

```
1. #include<stdio.h>
2. int addition ();
3. int main ()
4. {
5.     int result;
6.     int (*ptr)();
7.     ptr = &addition;
8.     result = (*ptr)();
9.     printf("The sum is %d",result);
10.}
11.int addition()
12.{
13.    int a, b;
14.    printf("Enter two numbers?");
15.    scanf("%d %d",&a,&b);
16.    return a+b;
17.}
```

Output

```
Enter two numbers?10 15
The sum is 25
```

Pointer to Array of functions in C

To understand the concept of an array of functions, we must understand the array of function. Basically, an array of the function is an array which contains the addresses of functions. In other words, the pointer to an array of functions is a pointer pointing to an array which contains the pointers to the functions. Consider the following example.

```
1. #include<stdio.h>
2. int show();
3. int showadd(int);
4. int (*arr[3])();
5. int (*(*ptr)[3])();
6.
7. int main ()
8. {
9.     int result1;
10.    arr[0] = show;
11.    arr[1] = showadd;
12.    ptr = &arr;
13.    result1 = (**ptr)();
14.    printf("printing the value returned by show : %d",result1);
15.    ((*ptr+1))(result1);
16.}
17.int show()
18.{
19.    int a = 65;
20.    return a++;
21.}
22.int showadd(int b)
23.{
24.    printf("\nAdding 90 to the value returned by show: %d",b+90);
25.}
```

Output

```
printing the value returned by show : 65
```

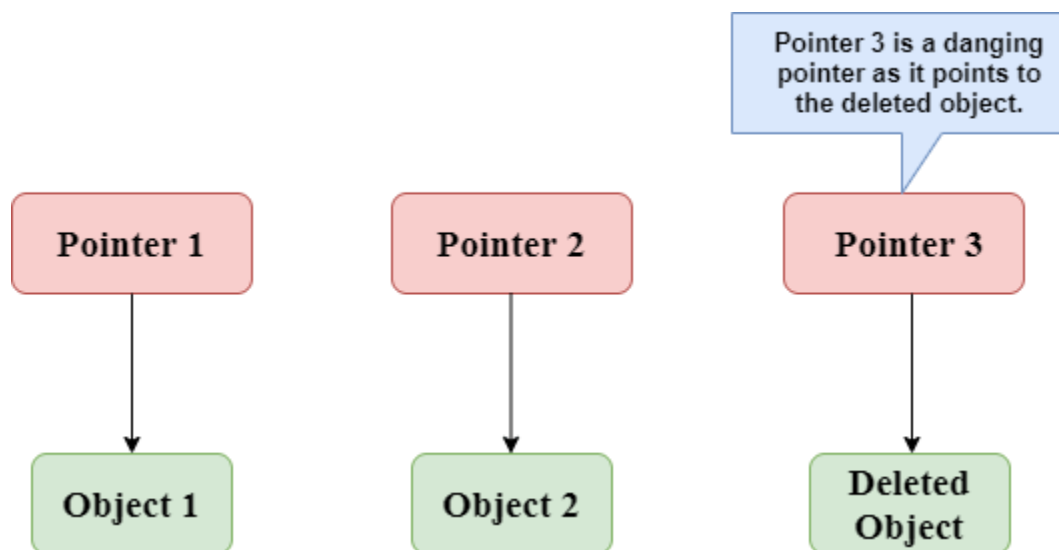
```
Adding 90 to the value returned by show: 155
```

Dangling Pointers in C

The most common bugs related to pointers and memory management is dangling/wild pointers. Sometimes the programmer fails to initialize the pointer with a valid address, then this type of initialized pointer is known as a dangling pointer in C.

Dangling pointer occurs at the time of the object destruction when the object is deleted or de-allocated from memory without modifying the value of the pointer. In this case, the pointer is pointing to the memory, which is de-allocated. The dangling pointer can point to the memory, which contains either the program code or the code of the operating system. If we assign the value to this pointer, then it overwrites the value of the program code or operating system instructions; in such cases, the program will show the undesirable result or may even crash. If the memory is re-allocated to some other process, then we dereference the dangling pointer will cause the segmentation faults.

Let's observe the following examples.



In the above figure, we can observe that the Pointer 3 is a dangling pointer. Pointer 1 and Pointer 2 are the pointers that point to the allocated objects, i.e., Object 1 and Object 2, respectively. Pointer 3 is a dangling pointer as it points to the de-allocated object.

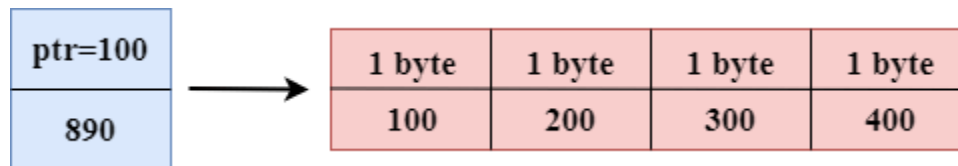
Let's understand the dangling pointer through some C programs.

Using free() function to de-allocate the memory.

```
1. #include <stdio.h>
2. int main()
3. {
4.     int *ptr=(int *)malloc(sizeof(int));
5.     int a=560;
6.     ptr=&a;
7.     free(ptr);
8.     return 0;
9. }
```

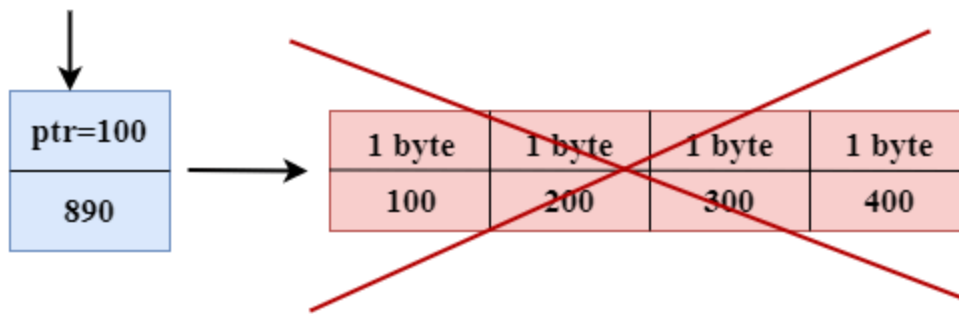
In the above code, we have created two variables, i.e., *ptr and a where 'ptr' is a pointer and 'a' is a integer variable. The *ptr is a pointer variable which is created with the help of malloc() function. As we know that malloc() function returns void, so we use int * to convert void pointer into int pointer.

The statement `int *ptr=(int *)malloc(sizeof(int));` will allocate the memory with 4 bytes shown in the below image:

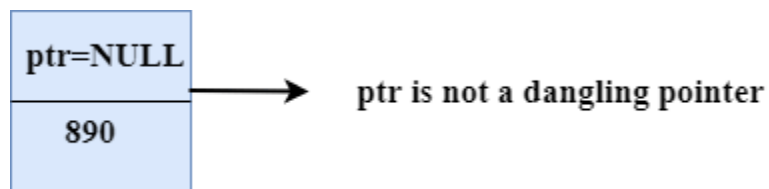


The statement `free(ptr)` de-allocates the memory as shown in the below image with a cross sign, and 'ptr' pointer becomes dangling as it is pointing to the de-allocated memory.

Dangling pointer



If we assign the NULL value to the 'ptr', then 'ptr' will not point to the deleted memory. Therefore, we can say that ptr is not a dangling pointer, as shown in the below image:



Variable goes out of the scope

When the variable goes out of the scope then the pointer pointing to the variable becomes a dangling pointer.

```
1. #include<stdio.h>
2. int main()
3. {
4.     char *str;
5.     {
6.         char a = 'A';
7.         str = &a;
8.     }
9.     // a falls out of scope
10.    // str is now a dangling pointer
11.    printf("%s", *str);
12.}
```

In the above code, we did the following steps:

- First, we declare the pointer variable named 'str'.
- In the inner scope, we declare a character variable. The str pointer contains the address of the variable 'a'.
- When the control comes out of the inner scope, 'a' variable will no longer be available, so str points to the de-allocated memory. It means that the str pointer becomes the dangling pointer.

Function call

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

Let's understand through an example.

```
1.  #include <stdio.h>
2.  int *fun(){
3.  int y=10;
4.  return &y;
5.  }
6.  int main()
7.  {
8.  int *p=fun();
9.  printf("%d", *p);
10. return 0;
11.}
```

In the above code, we did the following steps:

- First, we create the main() function in which we have declared 'p' pointer that contains the return value of the fun().

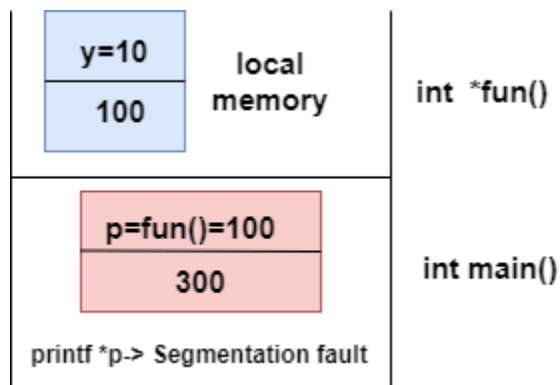
- When the fun() is called, then the control moves to the context of the int *fun(), the fun() returns the address of the 'y' variable.
- When control comes back to the context of the main() function, it means the variable 'y' is no longer available. Therefore, we can say that the 'p' pointer is a dangling pointer as it points to the de-allocated memory.

Output

```
Segmentation fault

...Program finished with exit code 139
Press ENTER to exit console.□
```

Let's represent the working of the above code diagrammatically.



Let's consider another example of a dangling pointer.

1. #include <stdio.h>
2. int *fun()
3. {
4. static int y=10;
5. return &y;

```

6. }
7. int main()
8. {
9.     int *p=fun();
10.    printf("%d", *p);
11.    return 0;
12.}

```

The above code is similar to the previous one but the only difference is that the variable 'y' is static. We know that static variable stores in the global memory.

Output

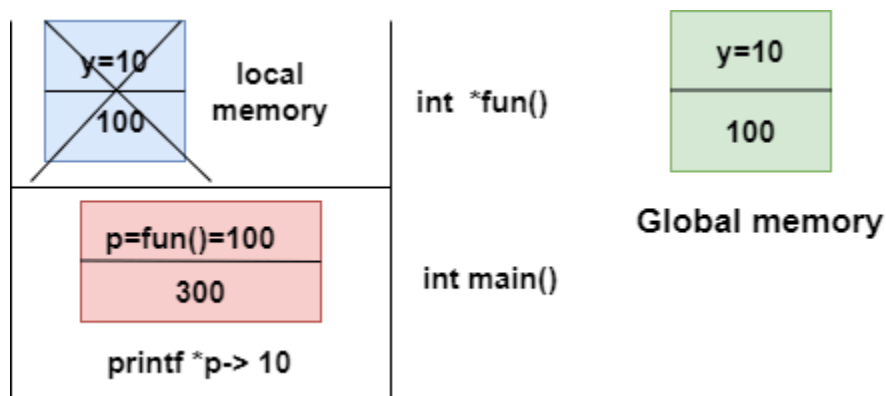
```

10

...Program finished with exit code 0
Press ENTER to exit console.

```

Now, we represent the working of the above code diagrammatically.



The above diagram shows the stack memory. First, the `fun()` function is called, then the control moves to the context of the `int *fun()`. As 'y' is a static variable, so it stores in the global memory; Its scope is available throughout the program. When the address value is returned, then the control comes back to the context of the `main()`. The pointer 'p' contains the address of 'y', i.e., 100. When we print the value of '*p', then it prints the

value of 'y', i.e., 10. Therefore, we can say that the pointer 'p' is not a dangling pointer as it contains the address of the variable which is stored in the global memory.

Avoiding Dangling Pointer Errors

The dangling pointer errors can be avoided by initializing the pointer to the NULL value. If we assign the NULL value to the pointer, then the pointer will not point to the de-allocated memory. Assigning NULL value to the pointer means that the pointer is not pointing to any memory location.

sizeof() operator in C

The sizeof() operator is commonly used in C. It determines the size of the expression or the data type specified in the number of char-sized storage units. The sizeof() operator contains a single operand which can be either an expression or a data typecast where the cast is data type enclosed within parenthesis. The data type cannot only be primitive data types such as integer or floating data types, but it can also be pointer data types and compound data types such as unions and structs.

Need of sizeof() operator

Mainly, programs know the storage size of the primitive data types. Though the storage size of the data type is constant, it varies when implemented in different platforms. For example, we dynamically allocate the array space by using sizeof() operator:

1. `int *ptr=malloc(10*sizeof(int));`

In the above example, we use the sizeof() operator, which is applied to the cast of type int. We use malloc() function to allocate the memory and returns the pointer which is pointing to this allocated memory. The memory space is equal to the number of bytes occupied by the int data type and multiplied by 10.

Note:

The output can vary on different machines such as on 32-bit operating system will show different output, and the 64-bit operating system will show the different outputs of the same data types.

The `sizeof()` operator behaves differently according to the type of the operand.

- Operand is a data type
- Operand is an expression

When operand is a data type.

```
1. #include <stdio.h>
2. int main()
3. {
4.     int x=89; // variable declaration.
5.     printf("size of the variable x is %d", sizeof(x)); // Displaying the size of ?x?
        variable.
6.     printf("\nsize of the integer data type is %d",sizeof(int)); //Displaying the size
        of integer data type.
7.     printf("\nsize of the character data type is %d",sizeof(char)); //Displaying the
        size of character data type.
8.
9.     printf("\nsize of the floating data type is %d",sizeof(float)); //Displaying the
        size of floating data type.
10. return 0;
11.}
```

In the above code, we are printing the size of different data types such as int, char, float with the help of `sizeof()` operator.

Output

```
size of the variable x is 4
size of the integer data type is 4
size of the character data type is 1
size of the floating data type is 4

...Program finished with exit code 0
Press ENTER to exit console.
```

When operand is an expression

1. `#include <stdio.h>`
2. `int main()`
3. `{`
4. `double i=78.0; //variable initialization.`
5. `float j=6.78; //variable initialization.`
6. `printf("size of (i+j) expression is : %d",sizeof(i+j)); //Displaying the size of the expression (i+j).`
7. `return 0;`
8. `}`

In the above code, we have created two variables 'i' and 'j' of type double and float respectively, and then we print the size of the expression by using `sizeof(i+j)` operator.

Output

```
size of (i+j) expression is : 8
```

const Pointer in C

Constant Pointers

A constant pointer in C cannot change the address of the variable to which it is pointing, i.e., the address will remain constant. Therefore, we can say that if a constant pointer is pointing to some variable, then it cannot point to any other variable.

Syntax of Constant Pointer

1. <type of pointer> ***const** <name of pointer>;

Declaration of a constant pointer is given below:

1. **int *const** ptr;

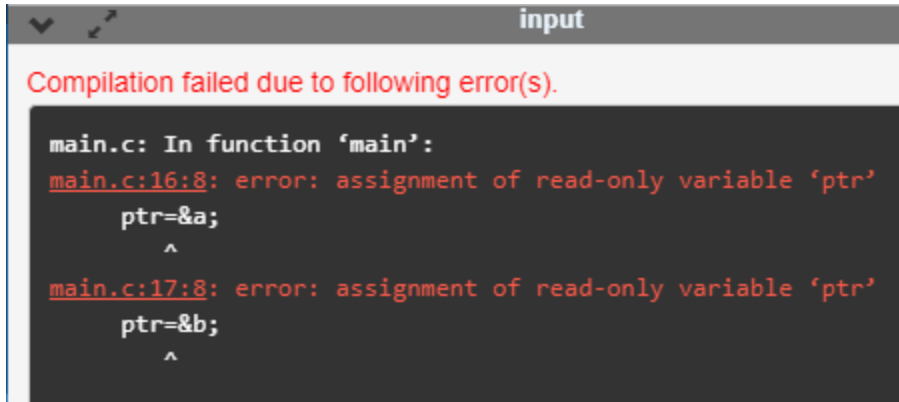
Let's understand the constant pointer through an example.

1. **#include <stdio.h>**
2. **int** main()
3. {
4. **int** a=1;
5. **int** b=2;
6. **int *const** ptr;
7. ptr=&a;
8. ptr=&b;
9. printf("Value of ptr is :%d",*ptr);
10. **return** 0;
- 11.}

In the above code:

- We declare two variables, i.e., a and b with values 1 and 2, respectively.
- We declare a constant pointer.
- First, we assign the address of variable 'a' to the pointer 'ptr'.
- Then, we assign the address of variable 'b' to the pointer 'ptr'.
- Lastly, we try to print the value of the variable pointed by the 'ptr'.

Output



```
input
Compilation failed due to following error(s).

main.c: In function 'main':
main.c:16:8: error: assignment of read-only variable 'ptr'
    ptr=&a;
    ^
main.c:17:8: error: assignment of read-only variable 'ptr'
    ptr=&b;
    ^
```

In the above output, we can observe that the above code produces the error "assignment of read-only variable 'ptr'". It means that the value of the variable 'ptr' which 'ptr' is holding cannot be changed. In the above code, we are changing the value of 'ptr' from &a to &b, which is not possible with constant pointers. Therefore, we can say that the constant pointer, which points to some variable, cannot point to another variable.

Pointer to Constant

A pointer to constant is a pointer through which the value of the variable that the pointer points cannot be changed. The address of these pointers can be changed, but the value of the variable that the pointer points cannot be changed.

Syntax of Pointer to Constant

1. **const** <type of pointer>* <name of pointer>

Declaration of a pointer to constant is given below:

1. **const int*** ptr;

Let's understand through an example.

- First, we write the code where we are changing the value of a pointer

```
1. #include <stdio.h>
2. int main()
3. {
4.     int a=100;
5.     int b=200;
6.     const int* ptr;
7.     ptr=&a;
8.     ptr=&b;
9.     printf("Value of ptr is :%u",ptr);
10.    return 0;
11.}
```

In the above code:

- We declare two variables, i.e., a and b with the values 100 and 200 respectively.
- We declare a pointer to constant.
- First, we assign the address of variable 'a' to the pointer 'ptr'.
- Then, we assign the address of variable 'b' to the pointer 'ptr'.
- Lastly, we try to print the value of 'ptr'.

Output

```
Value of ptr is :247760772
```

The above code runs successfully, and it shows the value of 'ptr' in the output.

- Now, we write the code in which we are changing the value of the variable to which the pointer points.

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



```
3. {
4.   int a=100;
5.   int b=200;
6.   const int* ptr;
7.   ptr=&b;
8.   *ptr=300;
9.   printf("Value of ptr is :%d",*ptr);
10.  return 0;
11.}
```

In the above code:

- We declare two variables, i.e., 'a' and 'b' with the values 100 and 200 respectively.
- We declare a pointer to constant.
- We assign the address of the variable 'b' to the pointer 'ptr'.
- Then, we try to modify the value of the variable 'b' through the pointer 'ptr'.
- Lastly, we try to print the value of the variable which is pointed by the pointer 'ptr'.

Output

```
main.c: In function 'main':
main.c:17:9: error: assignment of read-only location '*ptr'
    *ptr=300;
    ^
```

The above code shows the error "assignment of read-only location '*ptr'". This error means that we cannot change the value of the variable to which the pointer is pointing.

Constant Pointer to a Constant

A constant pointer to a constant is a pointer, which is a combination of the above two pointers. It can neither change the address of the variable to which it is pointing nor it can change the value placed at this address.

Syntax

1. `const <type of pointer>* const <name of the pointer>;`

Declaration for a constant pointer to a constant is given below:

1. `const int* const ptr;`

Let's understand through an example.

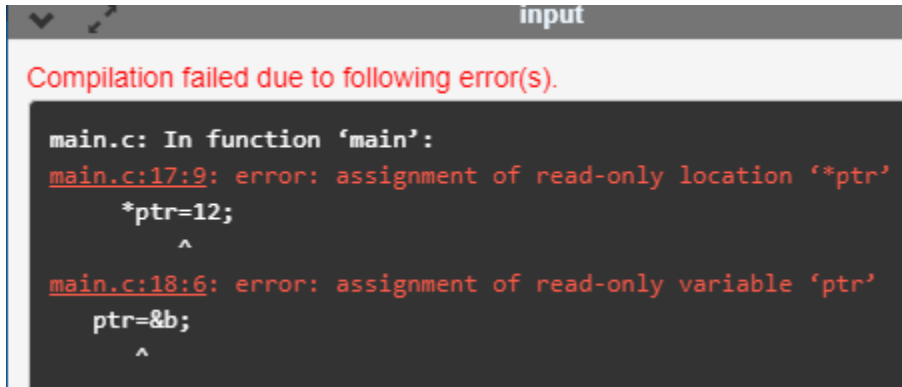
1. `#include <stdio.h>`
2. `int main()`
3. `{`
4. `int a=10;`
5. `int b=90;`
6. `const int* const ptr=&a;`
7. `*ptr=12;`
8. `ptr=&b;`
9. `printf("Value of ptr is :%d",*ptr);`
10. `return 0;`
11. `}`

In the above code:

- We declare two variables, i.e., 'a' and 'b' with the values 10 and 90, respectively.
- We declare a constant pointer to a constant and then assign the address of 'a'.
- We try to change the value of the variable 'a' through the pointer 'ptr'.
- Then we try to assign the address of variable 'b' to the pointer 'ptr'.

- Lastly, we print the value of the variable, which is pointed by the pointer 'ptr'.

Output



```
input
Compilation failed due to following error(s).

main.c: In function 'main':
main.c:17:9: error: assignment of read-only location '*ptr'
    *ptr=12;
    ^
main.c:18:6: error: assignment of read-only variable 'ptr'
    ptr=&b;
    ^
```

The above code shows the error "assignment of read-only location '*ptr'" and "assignment of read-only variable 'ptr'". Therefore, we conclude that the constant pointer to a constant can change neither address nor value, which is pointing by this pointer.

void pointer in C

Till now, we have studied that the address assigned to a pointer should be of the same type as specified in the pointer declaration. For example, if we declare the int pointer, then this int pointer cannot point to the float variable or some other type of variable, i.e., it can point to only int type variable. To overcome this problem, we use a pointer to void. A pointer to void means a generic pointer that can point to any data type. We can assign the address of any data type to the void pointer, and a void pointer can be assigned to any type of the pointer without performing any explicit typecasting.

Syntax of void pointer

1. **void** *pointer name;

Declaration of the void pointer is given below:

1. **void** *ptr;

In the above declaration, the void is the type of the pointer, and 'ptr' is the name of the pointer.

Let us consider some examples:

```
int i=9;    // integer variable initialization.

int *p;     // integer pointer declaration.

float *fp;  // floating pointer declaration.

void *ptr;  // void pointer declaration.

p=fp;       // incorrect.

fp=&i;       // incorrect

ptr=p;       // correct

ptr=fp;      // correct

ptr=&i;       // correct
```

Size of the void pointer in C

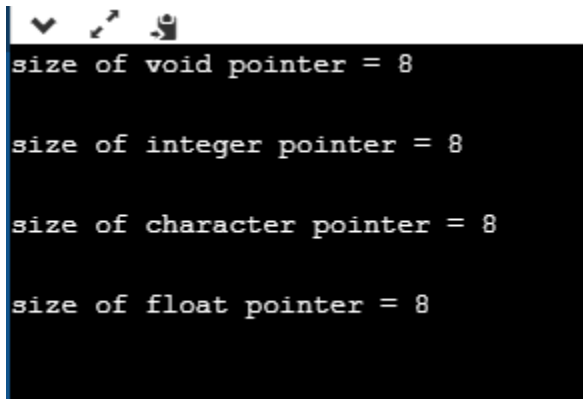
The size of the void pointer in C is the same as the size of the pointer of character type. According to C perception, the representation of a pointer to void is the same as the pointer of character type. The size of the pointer will vary depending on the platform that you are using.

Let's look at the below example:

1. `#include <stdio.h>`
2. `int main()`
3. `{`
4. `void *ptr = NULL; //void pointer`
5. `int *p = NULL; // integer pointer`
6. `char *cp = NULL; //character pointer`
7. `float *fp = NULL; //float pointer`
8. `//size of void pointer`
9. `printf("size of void pointer = %d\n\n", sizeof(ptr));`
10. `//size of integer pointer`

```
11. printf("size of integer pointer = %d\n\n",sizeof(p));
12. //size of character pointer
13. printf("size of character pointer = %d\n\n",sizeof(cp));
14. //size of float pointer
15. printf("size of float pointer = %d\n\n",sizeof(fp));
16. return 0;
17.}
```

Output



```
size of void pointer = 8

size of integer pointer = 8

size of character pointer = 8

size of float pointer = 8
```

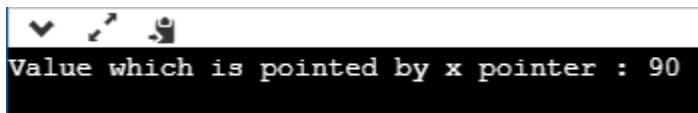
Advantages of void pointer

Following are the advantages of a void pointer:

- The malloc() and calloc() function return the void pointer, so these functions can be used to allocate the memory of any data type.

```
1. #include <stdio.h>
2. #include <malloc.h>
3. int main()
4. {
5.     int a=90;
6.
7.     int *x = (int*)malloc(sizeof(int));
8.     x=&a;
9.     printf("Value which is pointed by x pointer : %d",*x);
10.    return 0;
11.}
```

Output

A screenshot of a terminal window with a dark background. At the top, there is a toolbar with icons for a dropdown menu, a cursor, and a search icon. Below the toolbar, the text "Value which is pointed by x pointer : 90" is displayed in a light-colored monospace font.

- The void pointer in C can also be used to implement the generic functions in C.

Some important points related to void pointer are:

- Dereferencing a void pointer in C

The void pointer in C cannot be dereferenced directly. Let's see the below example.

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

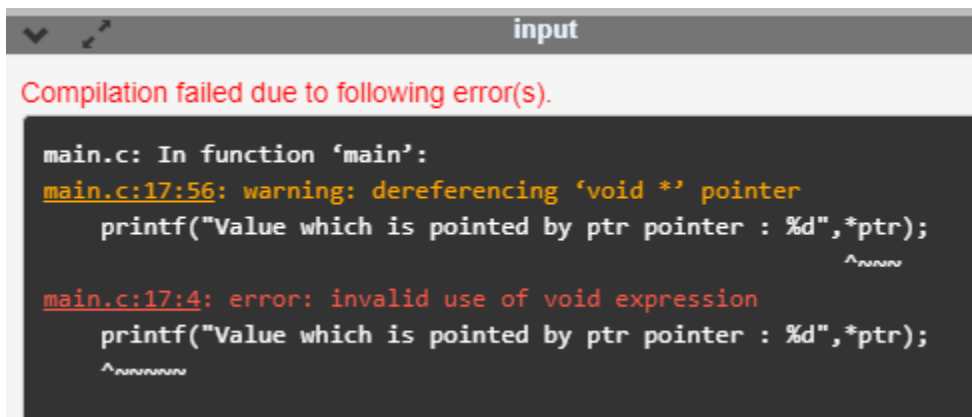
```

3. {
4.   int a=90;
5.   void *ptr;
6.   ptr=&a;
7.   printf("Value which is pointed by ptr pointer : %d",*ptr);
8.   return 0;
9. }

```

In the above code, *ptr is a void pointer which is pointing to the integer variable 'a'. As we already know that the void pointer cannot be dereferenced, so the above code will give the compile-time error because we are printing the value of the variable pointed by the pointer 'ptr' directly.

Output



The screenshot shows a compiler window titled 'input'. It displays a red message: 'Compilation failed due to following error(s)'. Below this, the compiler output is shown in a dark background with white and yellow text. The output indicates a warning and an error in 'main.c' at line 17. The warning is 'warning: dereferencing 'void *' pointer' at line 17:56, pointing to the asterisk in '*ptr' of the printf statement. The error is 'error: invalid use of void expression' at line 17:4, also pointing to the asterisk in '*ptr' of the printf statement.

```

main.c: In function 'main':
main.c:17:56: warning: dereferencing 'void *' pointer
    printf("Value which is pointed by ptr pointer : %d",*ptr);
                                                         ^~~~~~
main.c:17:4: error: invalid use of void expression
    printf("Value which is pointed by ptr pointer : %d",*ptr);
    ^~~~~~

```

Now, we rewrite the above code to remove the error.

1. `#include <stdio.h>`
2. `int main()`
3. `{`
4. `int a=90;`
5. `void *ptr;`
6. `ptr=&a;`
7. `printf("Value which is pointed by ptr pointer : %d",*(int*)ptr);`
8. `return 0;`
9. `}`

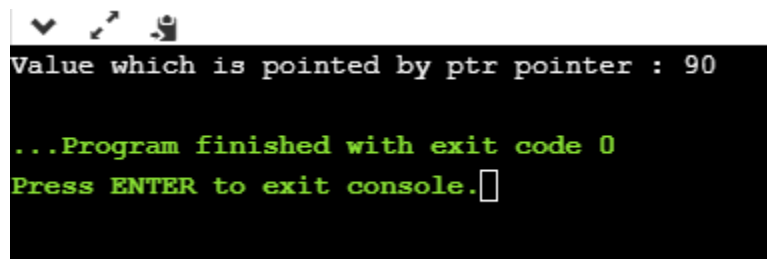
In the above code, we typecast the void pointer to the integer pointer by using the statement given below:

```
(int*)ptr;
```

Then, we print the value of the variable which is pointed by the void pointer 'ptr' by using the statement given below:

```
*(int*)ptr;
```

Output



```
Value which is pointed by ptr pointer : 90
...Program finished with exit code 0
Press ENTER to exit console.□
```

- Arithmetic operation on void pointers

We cannot apply the arithmetic operations on void pointers in C directly. We need to apply the proper typecasting so that we can perform the arithmetic operations on the void pointers.

Let's see the below example:


```

1. #include<stdio.h>
2. int main()
3. {
4.     float a[4]={6.1,2.3,7.8,9.0};
5.     void *ptr;
6.     ptr=a;
7.     for(int i=0;i<4;i++)
8.     {
9.         printf("%f",*ptr);
10.        ptr=ptr+1;    // Incorrect.
11.
12.    }}

```

The above code shows the compile-time error that "invalid use of void expression" as we cannot apply the arithmetic operations on void pointer directly, i.e., `ptr=ptr+1`.

Let's rewrite the above code to remove the error.

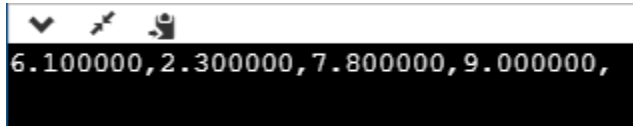
```

1. #include<stdio.h>
2. int main()
3. {
4.     float a[4]={6.1,2.3,7.8,9.0};
5.     void *ptr;
6.     ptr=a;
7.     for(int i=0;i<4;i++)
8.     {
9.         printf("%f",*((float*)ptr+i));
10.    }}

```

The above code runs successfully as we applied the proper casting to the void pointer, i.e., `(float*)ptr` and then we apply the arithmetic operation, i.e., `*((float*)ptr+i)`.

Output



```
6.100000, 2.300000, 7.800000, 9.000000,
```

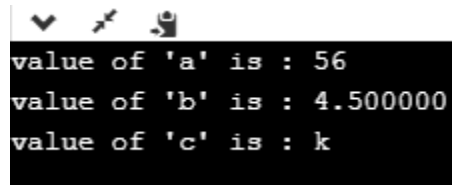
Why we use void pointers?

We use void pointers because of its reusability. Void pointers can store the object of any type, and we can retrieve the object of any type by using the indirection operator with proper typecasting.

Let's understand through an example.

```
1. #include<stdio.h>
2. int main()
3. {
4.     int a=56; // initialization of a integer variable 'a'.
5.     float b=4.5; // initialization of a float variable 'b'.
6.     char c='k'; // initialization of a char variable 'c'.
7.     void *ptr; // declaration of void pointer.
8.     // assigning the address of variable 'a'.
9.     ptr=&a;
10.    printf("value of 'a' is : %d",*((int*)ptr));
11.    // assigning the address of variable 'b'.
12.    ptr=&b;
13.    printf("\nvalue of 'b' is : %f",*((float*)ptr));
14.    // assigning the address of variable 'c'.
15.    ptr=&c;
16.    printf("\nvalue of 'c' is : %c",*((char*)ptr));
17.    return 0;
18.}
```

Output



```
value of 'a' is : 56  
value of 'b' is : 4.500000  
value of 'c' is : k
```

C dereference pointer

As we already know that "what is a pointer", a pointer is a variable that stores the address of another variable. The dereference operator is also known as an indirection operator, which is represented by (*). When indirection operator (*) is used with the pointer variable, then it is known as dereferencing a pointer. When we dereference a pointer, then the value of the variable pointed by this pointer will be returned.

Why we use dereferencing pointer?

Dereference a pointer is used because of the following reasons:

- It can be used to access or manipulate the data stored at the memory location, which is pointed by the pointer.
- Any operation applied to the dereferenced pointer will directly affect the value of the variable that it points to.

Let's observe the following steps to dereference a pointer.

- First, we declare the integer variable to which the pointer points.
 1. `int x = 9;`
- Now, we declare the integer pointer variable.
 1. `int *ptr;`
- After the declaration of an integer pointer variable, we store the address of 'x' variable to the pointer variable 'ptr'.

1. `ptr=&x;`

- We can change the value of 'x' variable by dereferencing a pointer 'ptr' as given below:

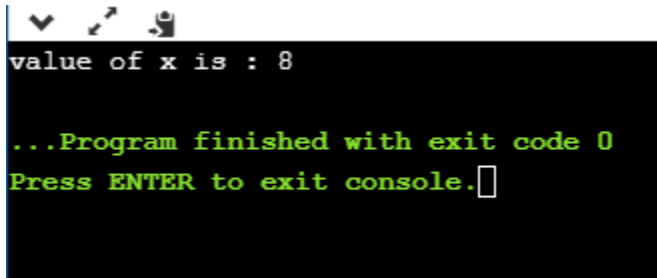
1. `*ptr=8;`

The above line changes the value of 'x' variable from 9 to 8 because 'ptr' points to the 'x' location and dereferencing of 'ptr', i.e., `*ptr=8` will update the value of x.

Let's combine all the above steps:

1. `#include <stdio.h>`
2. `int main()`
3. `{`
4. `int x=9;`
5. `int *ptr;`
6. `ptr=&x;`
7. `*ptr=8;`
8. `printf("value of x is : %d", x);`
9. `return 0;}`

Output



```
value of x is : 8

...Program finished with exit code 0
Press ENTER to exit console.
```

Let's consider another example.

1. `#include <stdio.h>`
2. `int main()`
3. `{`

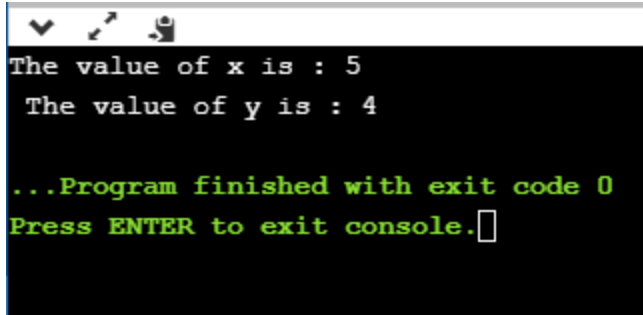
```
4.  int x=4;
5.  int y;
6.  int *ptr;
7.  ptr=&x;
8.  y=*ptr;
9.  *ptr=5;
10. printf("The value of x is : %d",x);
11. printf("\n The value of y is : %d",y);
12. return 0;
13.}
```

In the above code:

- We declare two variables 'x' and 'y' where 'x' is holding a '4' value.
- We declare a pointer variable 'ptr'.
- After the declaration of a pointer variable, we assign the address of the 'x' variable to the pointer 'ptr'.
- As we know that the 'ptr' contains the address of 'x' variable, so '*ptr' is the same as 'x'.
- We assign the value of 'x' to 'y' with the help of 'ptr' variable, i.e., y=*ptr instead of using the 'x' variable.

Note: According to us, if we change the value of 'x', then the value of 'y' will also get changed as the pointer 'ptr' holds the address of the 'x' variable. But this does not happen, as 'y' is storing the local copy of value '5'.

Output

A screenshot of a console window with a black background and white and green text. The text shows the output of a program: 'The value of x is : 5', 'The value of y is : 4', followed by '...Program finished with exit code 0' and 'Press ENTER to exit console.' with a cursor. The window has standard OS icons at the top.

```
The value of x is : 5
The value of y is : 4

...Program finished with exit code 0
Press ENTER to exit console.
```

Let's consider another scenario.

1. `#include <stdio.h>`
2. `int main()`
3. `{`
4. `int a=90;`
5. `int *ptr1,*ptr2;`
6. `ptr1=&a;`
7. `ptr2=&a;`
8. `*ptr1=7;`
9. `*ptr2=6;`
10. `printf("The value of a is : %d",a);`
11. `return 0;`
12. `}`

In the above code:

- First, we declare an 'a' variable.
- Then we declare two pointers, i.e., ptr1 and ptr2.
- Both the pointers contain the address of 'a' variable.
- We assign the '7' value to the *ptr1 and '6' to the *ptr2. The final value of 'a' would be '6'.

Note: If we have more than one pointer pointing to the same location, then the change made by one pointer will be the same as another pointer.

What is a Null Pointer?

A Null Pointer is a pointer that does not point to any memory location. It stores the base address of the segment. The null pointer basically stores the Null value while void is the type of the pointer.

A null pointer is a special reserved value which is defined in a stddef header file. Here, Null means that the pointer is referring to the 0th memory location.

If we do not have any address which is to be assigned to the pointer, then it is known as a null pointer. When a NULL value is assigned to the pointer, then it is considered as a Null pointer.

Applications of Null Pointer

Following are the applications of a Null pointer:

- It is used to initialize a pointer variable when the pointer does not point to a valid memory address.
- It is used to perform error handling with pointers before dereferencing the pointers.
- It is passed as a function argument and to return from a function when we do not want to pass the actual memory address.

Examples of Null Pointer

```
int *ptr=(int *)0;
```

```
float *ptr=(float *)0;
```

```
char *ptr=(char *)0;
```

```
double *ptr=(double *)0;
```

```
char *ptr='\0';
```

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

Let's look at the situations where we need to use the null pointer.

- When we do not assign any memory address to the pointer variable.

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

```
2. int main()
```

```
3. {
```

```
4.     int *ptr;
```

```
5.     printf("Address: %d", ptr); // printing the value of ptr.
```

```
6.     printf("Value: %d", *ptr); // dereferencing the illegal pointer
```

```
7.     return 0;
```

```
8. }
```

In the above code, we declare the pointer variable `*ptr`, but it does not contain the address of any variable. The dereferencing of the uninitialized pointer variable will show the compile-time error as it does not point any variable. According to the stack memory concept, the local variables of a function are stored in the stack, and if the variable does not contain any value, then it shows the garbage value. The above program shows some unpredictable results and causes the program to crash. Therefore, we can say that keeping an uninitialized pointer in a program can cause serious harm to the computer.

How to avoid the above situation?

We can avoid the above situation by using the Null pointer. A null pointer is a pointer pointing to the 0th memory location, which is a reserved memory and cannot be dereferenced.

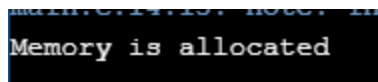

```
1. #include <stdio.h>
2. int main()
3. {
4.     int *ptr=NULL;
5.     if(ptr!=NULL)
6.     {
7.         printf("value of ptr is : %d",*ptr);
8.     }
9.     else
10.    {
11.        printf("Invalid pointer");
12.    }
13.    return 0;
14.}
```

In the above code, we create a pointer `*ptr` and assigns a `NULL` value to the pointer, which means that it does not point any variable. After creating a pointer variable, we add the condition in which we check whether the value of a pointer is null or not.

- When we use the malloc() function.

```
1. #include <stdio.h>
2. int main()
3. {
4.     int *ptr;
5.     ptr=(int*)malloc(4*sizeof(int));
6.     if(ptr==NULL)
7.     {
8.         printf("Memory is not allocated");
9.     }
10.    else
11.    {
12.        printf("Memory is allocated");
13.    }
14.    return 0;
15.}
```

In the above code, we use the library function, i.e., malloc(). As we know, that malloc() function allocates the memory; if malloc() function is not able to allocate the memory, then it returns the NULL pointer. Therefore, it is necessary to add the condition which will check whether the value of a pointer is null or not, if the value of a pointer is not null means that the memory is allocated.

A screenshot of a terminal window with a black background. The text 'Memory is allocated' is displayed in a light blue or cyan monospaced font.

Note: It is always a good practice that we assign a Null value to the pointer when we do not know the exact address of memory.

C Function Pointer

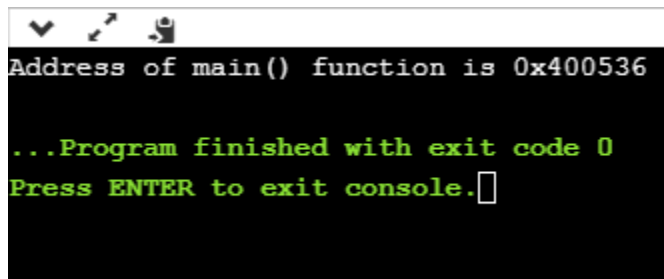
As we know that we can create a pointer of any data type such as int, char, float, we can also create a pointer pointing to a function. The code of a function always resides in memory, which means that the function has some address. We can get the address of memory by using the function pointer.

Let's see a simple example.

1. `#include <stdio.h>`
2. `int main()`
3. `{`
4. `printf("Address of main() function is %p",main);`
5. `return 0;`
6. `}`

The above code prints the address of main() function.

Output



```
Address of main() function is 0x400536

...Program finished with exit code 0
Press ENTER to exit console.
```

In the above output, we observe that the main() function has some address. Therefore, we conclude that every function has some address.

Declaration of a function pointer

Till now, we have seen that the functions have addresses, so we can create pointers that can contain these addresses, and hence can point them.

Syntax of function pointer

1. **return** type (*ptr_name)(type1, type2...);

For example:

1. **int** (*ip) (**int**);

In the above declaration, *ip is a pointer that points to a function which returns an int value and accepts an integer value as an argument.

1. **float** (*fp) (**float**);

In the above declaration, *fp is a pointer that points to a function that returns a float value and accepts a float value as an argument.

We can observe that the declaration of a function is similar to the declaration of a function pointer except that the pointer is preceded by a '*'. So, in the above declaration, fp is declared as a function rather than a pointer.

Till now, we have learnt how to declare the function pointer. Our next step is to assign the address of a function to the function pointer.

```
1. float (*fp) (int , int); // Declaration of a function pointer.  
2. float func(int , int ); // Declaration of function.  
3. fp = func; // Assigning address of func to the fp pointer.
```

In the above declaration, 'fp' pointer contains the address of the 'func' function.

Note: Declaration of a function is necessary before assigning the address of a function to the function pointer.

Calling a function through a function pointer

We already know how to call a function in the usual way. Now, we will see how to call a function using a function pointer.

Suppose we declare a function as given below:

1. **float** func(**int** , **int**); // Declaration of a function.

Calling an above function using a usual way is given below:

```
1. result = func(a , b);    // Calling a function using usual ways.
```

Calling a function using a function pointer is given below:

```
1. result = (*fp)( a , b);  // Calling a function using function pointer.
```

Or

```
1. result = fp(a , b);      // Calling a function using function pointer, and indirection  
operator can be removed.
```

The effect of calling a function by its name or function pointer is the same. If we are using the function pointer, we can omit the indirection operator as we did in the second case. Still, we use the indirection operator as it makes it clear to the user that we are using a function pointer.

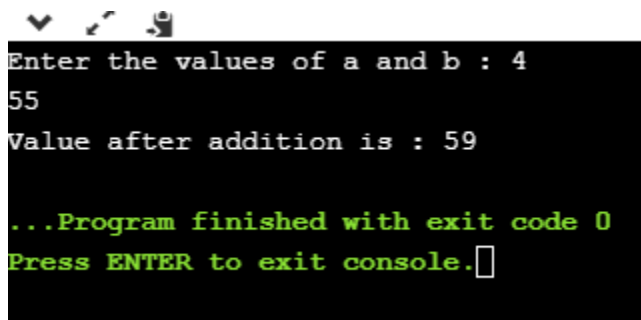
Let's understand the function pointer through an example.

```

1. #include <stdio.h>
2. int add(int,int);
3. int main()
4. {
5.     int a,b;
6.     int (*ip)(int,int);
7.     int result;
8.     printf("Enter the values of a and b : ");
9.     scanf("%d %d",&a,&b);
10.    ip=add;
11.    result=(*ip)(a,b);
12.    printf("Value after addition is : %d",result);
13.    return 0;
14.}
15.int add(int a,int b)
16.{
17.    int c=a+b;
18.    return c;
19.}

```

Output



```

Enter the values of a and b : 4
55
Value after addition is : 59

...Program finished with exit code 0
Press ENTER to exit console.

```

Passing a function's address as an argument to other function

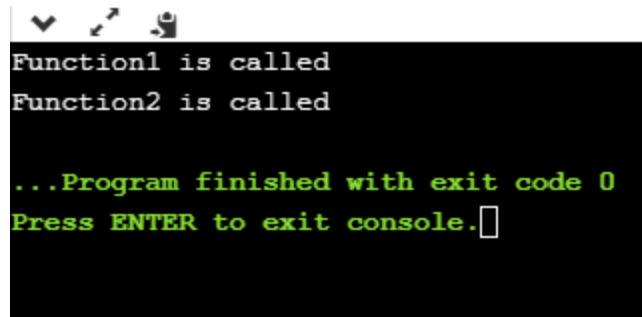
We can pass the function's address as an argument to other functions in the same way we send other arguments to the function.

Let's understand through an example.

```
1. include <stdio.h>
2. void func1(void (*ptr)());
3. void func2();
4. int main()
5. {
6.     func1(func2);
7.     return 0;
8. }
9. void func1(void (*ptr)())
10.{
11.    printf("Function1 is called");
12.    (*ptr)();
13.}
14.void func2()
15.{
16.    printf("\nFunction2 is called");
17.}
```

In the above code, we have created two functions, i.e., func1() and func2(). The func1() function contains the function pointer as an argument. In the main() method, the func1() method is called in which we pass the address of func2. When func1() function is called, 'ptr' contains the address of 'func2'. Inside the func1() function, we call the func2() function by dereferencing the pointer 'ptr' as it contains the address of func2.

Output

A screenshot of a console window with a black background and white and green text. At the top, there are three small icons: a checkmark, a cursor, and a document. The text in the console reads: "Function1 is called", "Function2 is called", "...Program finished with exit code 0", and "Press ENTER to exit console." followed by a cursor icon.

```
Function1 is called
Function2 is called

...Program finished with exit code 0
Press ENTER to exit console.
```

Array of Function Pointers

Function pointers are used in those applications where we do not know in advance which function will be called. In an array of function pointers, array takes the addresses of different functions, and the appropriate function will be called based on the index number.

Let's understand through an example.

1. `#include <stdio.h>`
2. `float add(float,int);`
3. `float sub(float,int);`
4. `float mul(float,int);`
5. `float div(float,int);`
6. `int main()`
7. `{`
8. `float x; // variable declaration.`
9. `int y;`
10. `float (*fp[4]) (float,int); // function pointer declaration.`
11. `fp[0]=add; // assigning addresses to the elements of an array of a
 function pointer.`
12. `fp[1]=sub;`
13. `fp[2]=mul;`
14. `fp[3]=div;`
15. `printf("Enter the values of x and y :");`


```

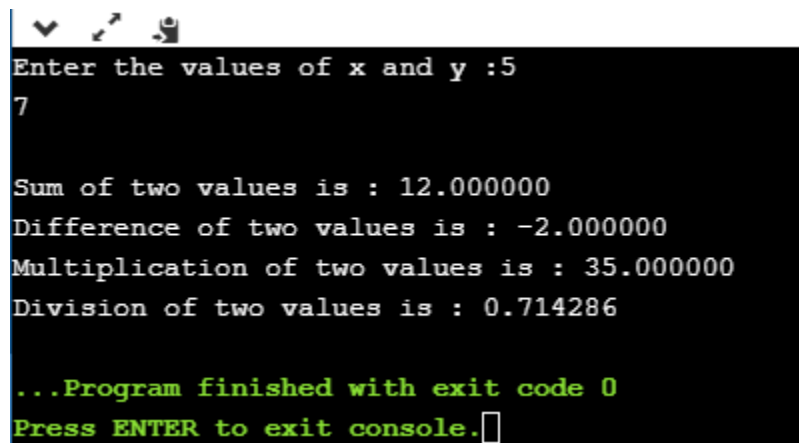
16. scanf("%f %d",&x,&y);
17. float r=(*fp[0]) (x,y);    // Calling add() function.
18. printf("\nSum of two values is : %f",r);
19. r=(*fp[1]) (x,y);          // Calling sub() function.
20. printf("\nDifference of two values is : %f",r);
21. r=(*fp[2]) (x,y);          // Calling sub() function.
22. printf("\nMultiplication of two values is : %f",r);
23. r=(*fp[3]) (x,y);          // Calling div() function.
24. printf("\nDivision of two values is : %f",r);
25. return 0;
26.}
27.
28.float add(float x,int y)
29.{
30. float a=x+y;
31. return a;
32.}
33.float sub(float x,int y)
34.{
35. float a=x-y;
36. return a;
37.}
38.float mul(float x,int y)
39.{
40. float a=x*y;
41. return a;
42.}
43.float div(float x,int y)
44.{

```

```
45. float a=x/y;
46. return a;
47.}
```

In the above code, we have created an array of function pointers that contain the addresses of four functions. After storing the addresses of functions in an array of function pointers, we call the functions using the function pointer.

Output



```
Enter the values of x and y :5
7

Sum of two values is : 12.000000
Difference of two values is : -2.000000
Multiplication of two values is : 35.000000
Division of two values is : 0.714286

...Program finished with exit code 0
Press ENTER to exit console.
```

Function pointer as argument in C

Till now, we have seen that in C programming, we can pass the variables as an argument to a function. We cannot pass the function as an argument to another function. But we can pass the reference of a function as a parameter by using a function pointer. This process is known as call by reference as the function parameter is passed as a pointer that holds the address of arguments. If any change made by the function using pointers, then it will also reflect the changes at the address of the passed variable.

Therefore, C programming allows you to create a pointer pointing to the function, which can be further passed as an argument to the function. We can create a function pointer as follows:

1. (type) (*pointer_name)(parameter);

In the above syntax, the type is the variable type which is returned by the function, *pointer_name is the function pointer, and the parameter is the list of the argument passed to the function.

Let's consider an example:

1. `float (*add)(); // this is a legal declaration for the function pointer`
2. `float *add(); // this is an illegal declaration for the function pointer`

A function pointer can also point to another function, or we can say that it holds the address of another function.

1. `float add (int a, int b); // function declaration`
2. `float (*a)(int, int); // declaration of a pointer to a function`
3. `a=add; // assigning address of add() to 'a' pointer`

In the above case, we have declared a function named as 'add'. We have also declared the function pointer (*a) which returns the floating-type value, and contains two parameters of integer type. Now, we can assign the address of add() function to the 'a' pointer as both are having the same return type(float), and the same type of arguments.

Now, 'a' is a pointer pointing to the add() function. We can call the add() function by using the pointer, i.e., 'a'. Let's see how we can do that:

1. `a(2, 3);`

The above statement calls the add() function by using pointer 'a', and two parameters are passed in 'a', i.e., 2 and 3.

Let's see a simple example of how we can pass the function pointer as a parameter.

1. `void display(void (*p)())`
2. `{`
3. `for(int i=1;i<=5;i++)`
4. `{`
5. `p(i);`
6. `}`

```

7. }
8. void print_numbers(int num)
9. {
10.  cout<<num;
11.}
12.int main()
13.{
14.  void (*p)(int);  // void function pointer declaration
15.  printf("The values are :");
16.  display(print_numbers);
17.  return 0;
18.}

```

In the above code,

- We have defined two functions named 'display()' and print_numbers().
- Inside the main() method, we have declared a function pointer named as (*p), and we call the display() function in which we pass the print_numbers() function.
- When the control goes to the display() function, then pointer *p contains the address of print_numbers() function. It means that we can call the print_numbers() function using function pointer *p.
- In the definition of display() function, we have defined a 'for' loop, and inside the for loop, we call the print_numbers() function using statement p(i). Here, p(i) means that print_numbers() function will be called on each iteration of i, and the value of 'i' gets printed.

Output

```
The values are :
1
2
3
4
5

...Program finished with exit code 0
Press ENTER to exit console.
```

Now, we will pass the function pointer as a argument in Quicksort function "qsort". It uses an algorithm that sorts an array.

1. `#include <stdio.h>`
2. `#include <stdlib.h>`
- 3.
4. `#include<string.h>`
5. `int compare(const int *p, const int *q);`
6. `int (*f)(const void *a, const void *b);`
7. `int main()`
8. `{`
9. `int a[]={4,7,6,1,3,2};`
10. `int num=sizeof(a)/sizeof(int);`
11. `f=&compare;`
12. `qsort(a, num, sizeof(int), (*f));`
13. `for(int i=0;i<num;i++)`
14. `{`
15. `printf("%d ",a[i]);`
16. `}`
- 17.
18. `}`

```
19.  
20.int compare(const int *p, const int *q)  
21.{  
22.    if (*p == *q)  
23.        return 0;  
24.    else if (*p < *q)  
25.        return -1;  
26.    else  
27.        return 1;  
28.}
```

In the above code,

- We have defined an array of integer type. After creating an array, we have calculated the size of an array by using the sizeof() operator, and stores the size in the num
- We define a compare() function, which compares all the elements in an array and arranges them in ascending order.
- We also have declared the function pointer, i.e., (*f), and stores the address of compare() function in (*f) by using the statement f=&compare.
- We call qsort() function in which we pass the array, size of the array, size of the element, and the comparison function. The comparison function, i.e., compare() will compare the array elements until the elements in an array get sorted in ascending order.

Output

```
main.c:19:6: warning: assignment from  
1 ,2 ,3 ,4 ,6 ,7 ,  
  
...Program finished with exit code 0  
Press ENTER to exit console.
```

Dynamic memory allocation in C

<https://www.geeksforgeeks.org/dynamic-memory-allocation-in-c-using-malloc-calloc-free-and-realloc/amp/>

The concept of dynamic memory allocation in c language *enables the C programmer to allocate memory at runtime*. Dynamic memory allocation in c language is possible by 4 functions of stdlib.h header file.

1. malloc()
2. calloc()
3. realloc()
4. free()

Before learning above functions, let's understand the difference between static memory allocation and dynamic memory allocation.

static memory allocation	dynamic memory allocation
memory is allocated at compile time.	memory is allocated at run time.
memory can't be increased while executing program.	memory can be increased while executing program.
used in array.	used in linked list.

Now let's have a quick look at the methods used for dynamic memory allocation.

malloc()	allocates single block of requested memory.
calloc()	allocates multiple block of requested memory.
realloc()	reallocates the memory occupied by malloc() or calloc() functions.
free()	frees the dynamically allocated memory.

malloc() function in C

The malloc() function allocates single block of requested memory.

It doesn't initialize memory at execution time, so it has garbage value initially.

It returns NULL if memory is not sufficient.

The syntax of malloc() function is given below:

1. `ptr=(cast-type*)malloc(byte-size)`

Let's see the example of malloc() function.

1. `#include<stdio.h>`
2. `#include<stdlib.h>`
3. `int main(){`
4. `int n,i,*ptr,sum=0;`
5. `printf("Enter number of elements: ");`
6. `scanf("%d",&n);`
7. `ptr=(int*)malloc(n*sizeof(int)); //memory allocated using malloc`
8. `if(ptr==NULL)`
9. `{`
10. `printf("Sorry! unable to allocate memory");`


```
11.    exit(0);
12. }
13. printf("Enter elements of array: ");
14. for(i=0;i<n;++i)
15. {
16.     scanf("%d",ptr+i);
17.     sum+=*(ptr+i);
18. }
19. printf("Sum=%d",sum);
20. free(ptr);
21. return 0;
22.}
```

Output

```
Enter elements of array: 3
Enter elements of array: 10
10
10
Sum=30
```

calloc() function in C

The calloc() function allocates multiple block of requested memory.

It initially initialize all bytes to zero.

It returns NULL if memory is not sufficient.

The syntax of calloc() function is given below:

```
1. ptr=(cast-type*)calloc(number, byte-size)
```

Let's see the example of calloc() function.

```
1. #include<stdio.h>
2. #include<stdlib.h>
3. int main(){
4.     int n,i,*ptr,sum=0;
5.     printf("Enter number of elements: ");
6.     scanf("%d",&n);
7.     ptr=(int*)calloc(n,sizeof(int)); //memory allocated using calloc
8.     if(ptr==NULL)
9.     {
10.        printf("Sorry! unable to allocate memory");
11.        exit(0);
12.    }
13.    printf("Enter elements of array: ");
14.    for(i=0;i<n;++i)
15.    {
16.        scanf("%d",ptr+i);
17.        sum+=*(ptr+i);
18.    }
19.    printf("Sum=%d",sum);
20.    free(ptr);
21.    return 0;
22.}
```

Output

```
Enter elements of array: 3
Enter elements of array: 10
10
10
Sum=30
```

realloc() function in C

If memory is not sufficient for malloc() or calloc(), you can reallocate the memory by realloc() function. In short, it changes the memory size.

Let's see the syntax of realloc() function.

1. `ptr=realloc(ptr, new-size)`

free() function in C

The memory occupied by malloc() or calloc() functions must be released by calling free() function. Otherwise, it will consume memory until program exit.

Let's see the syntax of free() function.

1. `free(ptr)`

C Strings

The string can be defined as the one-dimensional array of characters terminated by a null ('\0'). The character array or the string is used to manipulate text such as word or sentences. Each character in the array occupies one byte of memory, and the last character must always be 0. The termination character ('\0') is important in a string since it is the only way to identify where the string ends. When we define a string as `char s[10]`, the character `s[10]` is implicitly initialized with the null in the memory.

There are two ways to declare a string in c language.

1. By char array
2. By string literal

Let's see the example of declaring string by char array in C language.

1. `char ch[10]={ 'j', 'a', 'v', 'a', 't', 'p', 'o', 'i', 'n', 't', '\0' };`

As we know, array index starts from 0, so it will be represented as in the figure given below.

0	1	2	3	4	5	6	7	8	9	10
j	a	v	a	t	p	o	i	n	t	\0

While declaring string, size is not mandatory. So we can write the above code as given below:

```
1. char ch[]={ 'j', 'a', 'v', 'a', 't', 'p', 'o', 'i', 'n', 't', '\0'};
```

We can also define the string by the string literal in C language. For example:

```
1. char ch[]="javatpoint";
```

In such case, '\0' will be appended at the end of the string by the compiler.

Difference between char array and string literal

There are two main differences between char array and literal.

- We need to add the null character '\0' at the end of the array by ourself whereas, it is appended internally by the compiler in the case of the character array.
- The string literal cannot be reassigned to another set of characters whereas, we can reassign the characters of the array.

String Example in C

Let's see a simple example where a string is declared and being printed. The '%s' is used as a format specifier for the string in c language.

```
1. #include<stdio.h>
2. #include <string.h>
3. int main(){
```

```
4. char ch[11]={ 'j', 'a', 'v', 'a', 't', 'p', 'o', 'i', 'n', 't', '\0'};
5. char ch2[11]="javatpoint";
6.
7. printf("Char Array Value is: %s\n", ch);
8. printf("String Literal Value is: %s\n", ch2);
9. return 0;
10.}
```

Output

```
Char Array Value is: javatpoint
String Literal Value is: javatpoint
```

Traversing String

Traversing the string is one of the most important aspects in any of the programming languages. We may need to manipulate a very large text which can be done by traversing the text. Traversing string is somewhat different from the traversing an integer array. We need to know the length of the array to traverse an integer array, whereas we may use the null character in the case of string to identify the end of the string and terminate the loop.

Hence, there are two ways to traverse a string.

- By using the length of string
- By using the null character.

Let's discuss each one of them.

Using the length of string

Let's see an example of counting the number of vowels in a string.

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

```

2. void main ()
3. {
4.     char s[11] = "javatpoint";
5.     int i = 0;
6.     int count = 0;
7.     while(i<11)
8.     {
9.         if(s[i]=='a' || s[i] == 'e' || s[i] == 'i' || s[i] == 'u' || s[i] == 'o')
10.        {
11.            count ++;
12.        }
13.        i++;
14.    }
15.    printf("The number of vowels %d",count);
16.}

```

Output

The number of vowels 4



Using the null character

Let's see the same example of counting the number of vowels by using the null character.

```

1. #include<stdio.h>
2. void main ()
3. {
4.     char s[11] = "javatpoint";
5.     int i = 0;
6.     int count = 0;

```

```
7.  while(s[i] != NULL)
8.  {
9.      if(s[i]=='a' || s[i] == 'e' || s[i] == 'i' || s[i] == 'u' || s[i] == 'o')
10.     {
11.         count ++;
12.     }
13.     i++;
14. }
15. printf("The number of vowels %d",count);
16.}
```

Output

The number of vowels 4

Accepting string as the input

Till now, we have used scanf to accept the input from the user. However, it can also be used in the case of strings but with a different scenario. Consider the below code which stores the string while space is encountered.

```
1. #include<stdio.h>
2. void main ()
3. {
4.     char s[20];
5.     printf("Enter the string?");
6.     scanf("%s",s);
7.     printf("You entered %s",s);
8. }
```

Output

```
Enter the string?javatpoint is the best
You entered javatpoint
```

It is clear from the output that, the above code will not work for space separated strings. To make this code working for the space separated strings, the minor change required in the scanf function, i.e., instead of writing `scanf("%s",s)`, we must write: `scanf("%[^\n]s",s)` which instructs the compiler to store the string `s` while the new line (`\n`) is encountered. Let's consider the following example to store the space-separated strings.

1. `#include<stdio.h>`
2. `void main ()`
3. `{`
4. `char s[20];`
5. `printf("Enter the string?");`
6. `scanf("%[^\n]s",s);`
7. `printf("You entered %s",s);`
8. `}`

Output

```
Enter the string?javatpoint is the best
You entered javatpoint is the best
```

Here we must also notice that we do not need to use address of (&) operator in scanf to store a string since string `s` is an array of characters and the name of the array, i.e., `s` indicates the base address of the string (character array) therefore we need not use & with it.

Some important points

However, there are the following points which must be noticed while entering the strings by using scanf.

- The compiler doesn't perform bounds checking on the character array. Hence, there can be a case where the length of the string can exceed the dimension of the character array which may always overwrite some important data.
- Instead of using scanf, we may use gets() which is an inbuilt function defined in a header file string.h. The gets() is capable of receiving only one string at a time.

Pointers with strings

We have used pointers with the array, functions, and primitive data types so far. However, pointers can be used to point to the strings. There are various advantages of using pointers to point strings. Let us consider the following example to access the string via the pointer.

1. `#include<stdio.h>`
2. `void main ()`
3. `{`
4. `char s[11] = "javatpoint";`
5. `char *p = s; // pointer p is pointing to string s.`
6. `printf("%s",p); // the string javatpoint is printed if we print p.`
7. `}`

Output

javatpoint



`char s[11] = "javatpoint"`

Index	0	1	2	3	4	5	6	7	8	9	10
values	j	a	v	a	t	p	o	i	n	t	\0
Address	20	21	22	23	24	25	26	27	28	29	30
Variable	ptr	char *ptr = s									
Value	20										
Address	10										

As we know that string is an array of characters, the pointers can be used in the same way they were used with arrays. In the above example, p is declared as a pointer to the array of characters s. P affects similar to s since s is the base address of the string and treated as a pointer internally. However, we can not change the content of s or copy the content of s into another string directly. For this purpose, we need to use the pointers to store the strings. In the following example, we have shown the use of pointers to copy the content of a string into another.

1. `#include<stdio.h>`
2. `void main ()`
3. `{`
4. `char *p = "hello javatpoint";`
5. `printf("String p: %s\n",p);`
6. `char *q;`
7. `printf("copying the content of p into q...\n");`
8. `q = p;`
9. `printf("String q: %s\n",q);`
10. `}`

Output

```
String p: hello javatpoint
copying the content of p into q...
String q: hello javatpoint
```

Once a string is defined, it cannot be reassigned to another set of characters. However, using pointers, we can assign the set of characters to the string. Consider the following example.

1. `#include<stdio.h>`
2. `void main ()`
3. `{`
4. `char *p = "hello javatpoint";`
5. `printf("Before assigning: %s\n",p);`
6. `p = "hello";`
7. `printf("After assigning: %s\n",p);`
8. `}`

Output

```
Before assigning: hello javatpoint
After assigning: hello
```

C gets() and puts() functions

The `gets()` and `puts()` are declared in the header file `stdio.h`. Both the functions are involved in the input/output operations of the strings.

C gets() function

The `gets()` function enables the user to enter some characters followed by the enter key. All the characters entered by the user get stored in a character array. The null

character is added to the array to make it a string. The `gets()` allows the user to enter the space-separated strings. It returns the string entered by the user.

Declaration

1. `char[] gets(char[]);`

Reading string using `gets()`

1. `#include<stdio.h>`
2. `void main ()`
3. `{`
4. `char s[30];`
5. `printf("Enter the string? ");`
6. `gets(s);`
7. `printf("You entered %s",s);`
8. `}`

Output

```
Enter the string?  
javatpoint is the best  
You entered javatpoint is the best
```

The `gets()` function is risky to use since it doesn't perform any array bound checking and keep reading the characters until the new line (enter) is encountered. It suffers from buffer overflow, which can be avoided by using `fgets()`. The `fgets()` makes sure that not more than the maximum limit of characters are read. Consider the following example.

1. `#include<stdio.h>`
2. `void main()`
3. `{`
4. `char str[20];`
5. `printf("Enter the string? ");`

6. `fgets(str, 20, stdin);`
7. `printf("%s", str);`
8. `}`

Output

Enter the string? javatpoint is the best website
javatpoint is the b



C puts() function

The `puts()` function is very much similar to `printf()` function. The `puts()` function is used to print the string on the console which is previously read by using `gets()` or `scanf()` function. The `puts()` function returns an integer value representing the number of characters being printed on the console. Since, it prints an additional newline character with the string, which moves the cursor to the new line on the console, the integer value returned by `puts()` will always be equal to the number of characters present in the string plus 1.

Declaration

1. `int puts(char[])`

Let's see an example to read a string using `gets()` and print it on the console using `puts()`.

1. `#include<stdio.h>`
2. `#include <string.h>`
3. `int main(){`
4. `char name[50];`
5. `printf("Enter your name: ");`
6. `gets(name); //reads string from user`
7. `printf("Your name is: ");`
8. `puts(name); //displays string`
9. `return 0;`

10.}

Output:

Enter your name: Sonoo Jaiswal

Your name is: Sonoo Jaiswal

C String Functions

There are many important string functions defined in "string.h" library.

No.	Function	Description
1)	strlen(string_name)	returns the length of string name.
2)	strcpy(destination, source)	copies the contents of source string to destination string.
3)	strcat(first_string, second_string)	concatenates or joins first string with second string. The result of the string is stored in first string.
4)	strcmp(first_string, second_string)	compares the first string with second string. If both strings are same, it returns 0.
5)	strrev(string)	returns reverse string.
6)	strlwr(string)	returns string characters in lowercase.
7)	strupr(string)	returns string characters in uppercase.

Next → ← Prev

C String Length: strlen() function

The strlen() function returns the length of the given string. It doesn't count null character '\0'.

```
1. #include<stdio.h>
2. #include <string.h>
3. int main(){
4.   char ch[20]={ 'j', 'a', 'v', 'a', 't', 'p', 'o', 'i', 'n', 't', '\0' };
5.   printf("Length of string is: %d",strlen(ch));
6.   return 0;
7. }
```

Output:

Length of string is: 10

C Copy String: strcpy()

The strcpy(destination, source) function copies the source string in destination.

```
1. #include<stdio.h>
2. #include <string.h>
3. int main(){
4.   char ch[20]={ 'j', 'a', 'v', 'a', 't', 'p', 'o', 'i', 'n', 't', '\0' };
5.   char ch2[20];
6.   strcpy(ch2,ch);
7.   printf("Value of second string is: %s",ch2);
8.   return 0;
9. }
```

Output:

Value of second_string is: javatpoint

C String Concatenation: strcat()

The `strcat(first_string, second_string)` function concatenates two strings and result is returned to `first_string`.

```
1. #include<stdio.h>
2. #include <string.h>
3. int main(){
4.     char ch[10]={'h', 'e', 'l', 'l', 'o', '\0'};
5.     char ch2[10]={'c', '\0'};
6.     strcat(ch,ch2);
7.     printf("Value of first string is: %s",ch);
8.     return 0;
9. }
```

Output:

Value of first string is: helloc

C Compare String: strcmp()

The `strcmp(first_string, second_string)` function compares two string and returns 0 if both strings are equal.

Here, we are using `gets()` function which reads string from the console.

```
1. #include<stdio.h>
2. #include <string.h>
3. int main(){
4.     char str1[20],str2[20];
```



```

5.  printf("Enter 1st string: ");
6.  gets(str1); //reads string from console
7.  printf("Enter 2nd string: ");
8.  gets(str2);
9.  if(strcmp(str1,str2)==0)
10.   printf("Strings are equal");
11. else
12.   printf("Strings are not equal");
13. return 0;
14.}

```

Output:

```

Enter 1st string: hello
Enter 2nd string: hello
Strings are equal

```

C Reverse String: strrev()

The strrev(string) function returns reverse of the given string. Let's see a simple example of strrev() function.

```

1. #include<stdio.h>
2. #include <string.h>
3. int main(){
4.   char str[20];
5.   printf("Enter string: ");
6.   gets(str); //reads string from console
7.   printf("String is: %s",str);
8.   printf("\nReverse String is: %s",strrev(str));
9.   return 0;

```

```
10.}
```

Output:

```
Enter string: javatpoint
```

```
String is: javatpoint
```

```
Reverse String is: tniopTava
```

C String Lowercase: strlwr()

The strlwr(string) function returns string characters in lowercase. Let's see a simple example of strlwr() function.

```
1. #include<stdio.h>
2. #include <string.h>
3. int main(){
4.     char str[20];
5.     printf("Enter string: "); //VANI
6.     gets(str); //reads string from console
7.     printf("String is: %s",str);
8.     printf("\nLower String is: %s",strlwr(str)); //vani
9.     return 0;
10.}
```

Output:

```
Enter string: JAVATpoint
```

```
String is: JAVATpoint
```

```
Lower String is: javatpoint
```

C String Uppercase:strupr()

The `strupr(string)` function returns string characters in uppercase. Let's see a simple example of `strupr()` function.

```
1. #include<stdio.h>
2. #include <string.h>
3. int main(){
4.     char str[20];
5.     printf("Enter string: ");
6.     gets(str);//reads string from console   vaniPatil
7.     printf("String is: %s",str);    //vaniPatil
8.     printf("\nUpper String is: %s",strupr(str));    //VANIPATIL
9.     return 0;
10.}
```

Output:

```
Enter string: javatpoint
String is: javatpoint
Upper String is: JAVATPOINT
```

C String strstr()

The `strstr()` function returns pointer to the first occurrence of the matched string in the given string. It is used to return substring from first match till the last character.

Syntax:

```
1. char *strstr(const char *string, const char *match)
```

String strstr() parameters

string: It represents the full string from where substring will be searched.

match: It represents the substring to be searched in the full string.

String strstr() example

```
1. #include<stdio.h>
2. #include <string.h>
3. int main(){
4.     char str[100]="this is javatpoint with c and java";
5.     char *sub;
6.     sub=strstr(str,"java");
7.     printf("\nSubstring is: %s",sub);
8.     return 0;
9. }
```

Output:

javatpoint with c and java

C Math

C Programming allows us to perform mathematical operations through the functions defined in <math.h> header file. The <math.h> header file contains various methods for performing mathematical operations such as sqrt(), pow(), ceil(), floor() etc.

C Math Functions

There are various methods in math.h header file. The commonly used functions of math.h header file are given below.

N	Function	Description
0.		

1)	<code>ceil(number)</code>	rounds up the given number. It returns the integer value which is greater than or equal to given number.
2)	<code>floor(number)</code>	rounds down the given number. It returns the integer value which is less than or equal to given number.
3)	<code>sqrt(number)</code>	returns the square root of given number.
4)	<code>pow(base, exponent)</code>	returns the power of given number.
5)	<code>abs(number)</code>	returns the absolute value of given number.

C Math Example

Let's see a simple example of math functions found in `math.h` header file.

```

1. #include<stdio.h>
2. #include <math.h>
3. int main(){
4. printf("\n%f",ceil(3.6));
5. printf("\n%f",ceil(3.3));
6. printf("\n%f",floor(3.6));
7. printf("\n%f",floor(3.2));
8. printf("\n%f",sqrt(16));
9. printf("\n%f",sqrt(7));
10.printf("\n%f",pow(2,4));
11.printf("\n%f",pow(3,3));
12.printf("\n%d",abs(-12));
13. return 0;
14.}
```

Output:

```
4.000000
4.000000
3.000000
3.000000
4.000000
2.645751
16.000000
27.000000
```

```
12
```

C Structure

Why use structure?

In C, there are cases where we need to store multiple attributes of an entity. It is not necessary that an entity has all the information of one type only. It can have different attributes of different data types. For example, an entity Student may have its name (string), roll number (int), marks (float). To store such type of information regarding an entity student, we have the following approaches:

- Construct individual arrays for storing names, roll numbers, and marks.
- Use a special data structure to store the collection of different data types.

Let's look at the first approach in detail.

1. `#include<stdio.h>`
2. `void main ()`
3. `{`
4. `char names[2][10],dummy; // 2-dimensionaal character array names is used to store the names of the students`
5. `int roll_numbers[2],i;`

```

6.  float marks[2];
7.  for (i=0;i<3;i++)
8.  {
9.
10.  printf("Enter the name, roll number, and marks of the student %d",i+1);
11.  scanf("%s %d %f",&names[i],&roll_numbers[i],&marks[i]);
12.  scanf("%c",&dummy); // enter will be stored into dummy character at each
    iteration
13. }
14. printf("Printing the Student details ...\n");
15. for (i=0;i<3;i++)
16. {
17.  printf("%s %d %f\n",names[i],roll_numbers[i],marks[i]);
18. }
19.}

```

Output

```

Enter the name, roll number, and marks of the student 1Arun 90 91
Enter the name, roll number, and marks of the student 2Varun 91 56
Enter the name, roll number, and marks of the student 3Sham 89 69

Printing the Student details...
Arun 90 91.000000
Varun 91 56.000000
Sham 89 69.000000

```

The above program may fulfill our requirement of storing the information of an entity student. However, the program is very complex, and the complexity increase with the amount of the input. The elements of each of the array are stored contiguously, but all the arrays may not be stored contiguously in the memory. C provides you with an additional and simpler approach where you can use a special data structure, i.e.,

structure, in which, you can group all the information of different data type regarding an entity.

What is Structure

Structure in c is a user-defined data type that enables us to store the collection of different data types. Each element of a structure is called a member. Structures can simulate the use of classes and templates as it can store various information

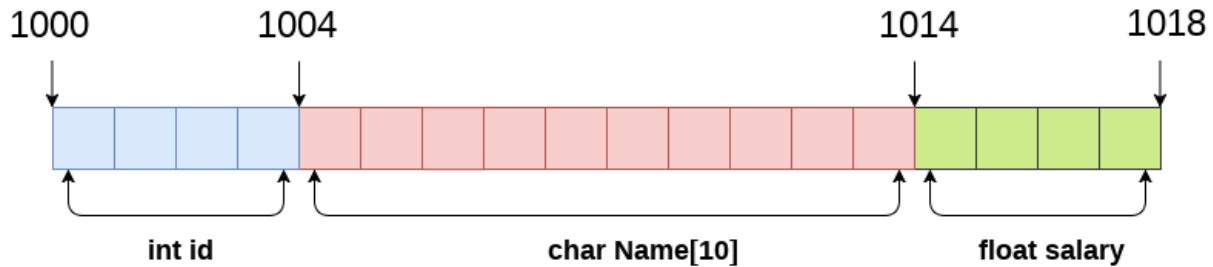
The `struct` keyword is used to define the structure. Let's see the syntax to define the structure in c.

```
1. struct structure_name
2. {
3.     data_type member1;
4.     data_type member2;
5.     .
6.     .
7.     data_type memberN;
8. };
```

Let's see the example to define a structure for an entity employee in c.

```
1. struct employee
2. { int id;
3.     char name[10];
4.     float salary;
5. };
```

The following image shows the memory allocation of the structure employee that is defined in the above example.



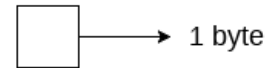
```

struct Employee
{
    int id;
    char Name[10];
    float salary;
} emp;

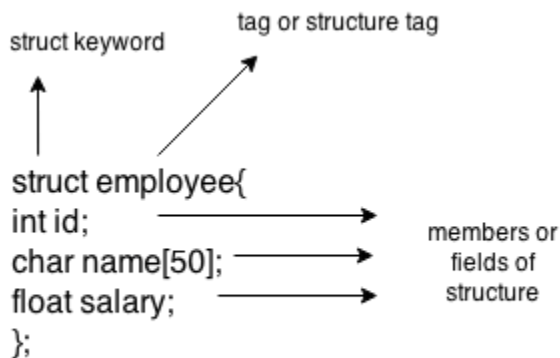
```

`sizeof (emp) = 4 + 10 + 4 = 18 bytes`

where;
`sizeof (int) = 4 byte`
`sizeof (char) = 1 byte`
`sizeof (float) = 4 byte`



Here, struct is the keyword; employee is the name of the structure; id, name, and salary are the members or fields of the structure. Let's understand it by the diagram given below:



JavaTpoint.com

Declaring structure variable

We can declare a variable for the structure so that we can access the member of the structure easily. There are two ways to declare structure variable:

1. By struct keyword within main() function
2. By declaring a variable at the time of defining the structure.

1st way:

Let's see the example to declare the structure variable by struct keyword. It should be declared within the main function.

1. **struct** employee
2. { **int** id;
3. **char** name[50];
4. **float** salary;
5. };

Now write given code inside the main() function.

1. **struct** employee e1, e2;

The variables e1 and e2 can be used to access the values stored in the structure. Here, e1 and e2 can be treated in the same way as the objects in **C++** and **Java**.

2nd way:

Let's see another way to declare variable at the time of defining the structure.

1. **struct** employee
2. { **int** id;
3. **char** name[50];
4. **float** salary;
5. }e1,e2;

Which approach is good

If number of variables are not fixed, use the 1st approach. It provides you the flexibility to declare the structure variable many times.

If no. of variables are fixed, use 2nd approach. It saves your code to declare a variable in main() function.

Accessing members of the structure

There are two ways to access structure members:

1. By . (member or dot operator)
2. By -> (structure pointer operator)

Let's see the code to access the *id* member of *p1* variable by. (member) operator.

1. `p1.id`

C Structure example

Let's see a simple example of structure in C language.

1. `#include<stdio.h>`
2. `#include <string.h>`
3. `struct employee`
4. `{ int id;`
5. `char name[50];`
6. `}e1; //declaring e1 variable for structure`
7. `int main()`
8. `{`
9. `//store first employee information`
10. `e1.id=101;`
11. `strcpy(e1.name, "Sonoo Jaisbwal");//copying string into char array`
12. `//printing first employee information`
13. `printf("employee 1 id : %d\n", e1.id);`
14. `printf("employee 1 name : %s\n", e1.name);`
15. `return 0;`
16. `}`

Output:

employee 1 id : 101

employee 1 name : Sonoo Jaiswal

Let's see another example of the structure in **C language** to store many employees information.

```
1. #include<stdio.h>
2. #include <string.h>
3. struct employee
4. { int id;
5.   char name[50];
6.   float salary;
7. }e1,e2; //declaring e1 and e2 variables for structure
8. int main()
9. {
10. //store first employee information
11. e1.id=101;
12. strcpy(e1.name, "Sonoo Jaiswal");//copying string into char array
13. e1.salary=56000;
14.
15. //store second employee information
16. e2.id=102;
17. strcpy(e2.name, "James Bond");
18. e2.salary=126000;
19.
20. //printing first employee information
21. printf( "employee 1 id : %d\n", e1.id);
22. printf( "employee 1 name : %s\n", e1.name);
23. printf( "employee 1 salary : %f\n", e1.salary);
24.
```

```

25. //printing second employee information
26. printf( "employee 2 id : %d\n", e2.id);
27. printf( "employee 2 name : %s\n", e2.name);
28. printf( "employee 2 salary : %f\n", e2.salary);
29. return 0;
30.}

```

Output:

```

employee 1 id : 101
employee 1 name : Sonoo Jaiswal
employee 1 salary : 56000.000000
employee 2 id : 102
employee 2 name : James Bond
employee 2 salary : 126000.000000

```

typedef in C

The typedef is a keyword used in C programming to provide some meaningful names to the already existing variable in the **C program**. It behaves similarly as we define the alias for the commands. In short, we can say that this keyword is used to redefine the name of an already existing variable.

Syntax of typedef

1. **typedef** <existing_name> <alias_name>

In the above syntax, 'existing_name' is the name of an already existing variable while 'alias name' is another name given to the existing variable.

For example, suppose we want to create a variable of type unsigned int, then it becomes a tedious task if we want to declare multiple variables of this type. To overcome the problem, we use a typedef keyword.

1. **typedef** unsigned **int** unit;

In the above statements, we have declared the unit variable of type unsigned int by using a typedef keyword.

Now, we can create the variables of type unsigned int by writing the following statement:

1. unit a, b;

instead of writing:

1. unsigned int a, b;

Till now, we have observed that the typedef keyword provides a nice shortcut by providing an alternative name for an already existing variable. This keyword is useful when we are dealing with the long data type especially, structure declarations.

Let's understand through a simple example.

1. #include <stdio.h>
2. int main()
3. {
4. typedef unsigned int unit;
5. unit i,j;
6. i=10;
7. j=20;
8. printf("Value of i is :%d",i);
9. printf("\nValue of j is :%d",j);
10. return 0;
11. }

Output

Value of i is :10

Value of j is :20

Using typedef with structures

Consider the below structure declaration:

1. **struct** student
2. {
3. **char** name[20];
4. **int** age;
5. };
6. **struct** student s1;

In the above structure declaration, we have created the variable of student type by writing the following statement:

1. **struct** student s1;

The above statement shows the creation of a variable, i.e., s1, but the statement is quite big. To avoid such a big statement, we use the typedef keyword to create the variable of type student.

1. **struct** student
2. {
3. **char** name[20];
4. **int** age;
5. };
6. **typedef struct** student stud;
7. stud s1, s2;

In the above statement, we have declared the variable stud of type struct student. Now, we can use the stud variable in a program to create the variables of type struct student.

The above typedef can be written as:

1. **typedef struct** student
2. {
3. **char** name[20];
4. **int** age;
5. } stud;
6. stud s1,s2;

From the above declarations, we conclude that typedef keyword reduces the length of the code and complexity of data types. It also helps in understanding the program.

Let's see another example where we typedef the structure declaration.

1. **#include <stdio.h>**
2. **typedef struct** student
3. {
4. **char** name[20];
5. **int** age;
6. }stud;
7. **int** main()
8. {
9. stud s1;
- 10.**printf("Enter the details of student s1: ");**
- 11.**printf("\nEnter the name of the student:");**
- 12.**scanf("%s",&s1.name);**
- 13.**printf("\nEnter the age of student:");**
- 14.**scanf("%d",&s1.age);**
- 15.**printf("\n Name of the student is : %s", s1.name);**
- 16.**printf("\n Age of the student is : %d", s1.age);**
- 17.**return 0;**
- 18.}

Output

```
Enter the details of student s1:  
Enter the name of the student: Peter  
Enter the age of student: 28  
Name of the student is : Peter  
Age of the student is : 28
```

Using typedef with pointers

We can also provide another name or alias name to the pointer variables with the help of the typedef.

For example, we normally declare a pointer, as shown below:

```
1. int* ptr;
```

We can rename the above pointer variable as given below:

```
1. typedef int* ptr;
```

In the above statement, we have declared the variable of type int*. Now, we can create the variable of type int* by simply using the 'ptr' variable as shown in the below statement:

```
1. ptr p1, p2 ;
```

In the above statement, p1 and p2 are the variables of type 'ptr'.

C Array of Structures

Why use an array of structures?

Consider a case, where we need to store the data of 5 students. We can store it by using the structure as given below.

```
1. #include<stdio.h>
2. struct student
3. {
4.     char name[20];
5.     int id;
6.     float marks;
7. };
8. void main()
9. {
10.    struct student s1,s2,s3;
11.    int dummy;
12.    printf("Enter the name, id, and marks of student 1 ");
13.    scanf("%s %d %f",s1.name,&s1.id,&s1.marks);
14.    scanf("%c",&dummy);
15.    printf("Enter the name, id, and marks of student 2 ");
16.    scanf("%s %d %f",s2.name,&s2.id,&s2.marks);
17.    scanf("%c",&dummy);
18.    printf("Enter the name, id, and marks of student 3 ");
19.    scanf("%s %d %f",s3.name,&s3.id,&s3.marks);
20.    scanf("%c",&dummy);
21.    printf("Printing the details....\n");
22.    printf("%s %d %f\n",s1.name,s1.id,s1.marks);
23.    printf("%s %d %f\n",s2.name,s2.id,s2.marks);
24.    printf("%s %d %f\n",s3.name,s3.id,s3.marks);
25.}
```

Output

```
Enter the name, id, and marks of student 1 James 90 90
Enter the name, id, and marks of student 2 Adams 90 90
```

```

Enter the name, id, and marks of student 3 Nick 90 90
Printing the details....
James 90 90.000000
Adams 90 90.000000
Nick 90 90.000000

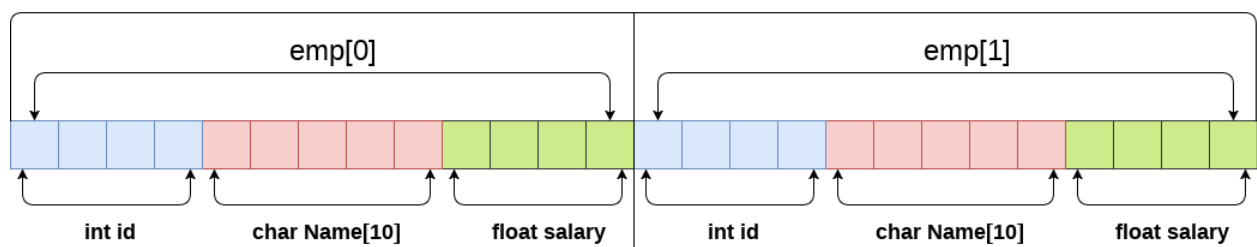
```

In the above program, we have stored data of 3 students in the structure. However, the complexity of the program will be increased if there are 20 students. In that case, we will have to declare 20 different structure variables and store them one by one. This will always be tough since we will have to declare a variable every time we add a student. Remembering the name of all the variables is also a very tricky task. However, C enables us to declare an array of structures by using which, we can avoid declaring the different structure variables; instead we can make a collection containing all the structures that store the information of different entities.

Array of Structures in C

An array of structures in C can be defined as the collection of multiple structures variables where each variable contains information about different entities. The array of structures in C are used to store information about multiple entities of different data types. The array of structures is also known as the collection of structures.

Array of structures



```

struct employee
{
    int id;
    char name[5];
    float salary;
};
struct employee emp[2];

```

`sizeof (emp) = 4 + 5 + 4 = 13 bytes`

`sizeof (emp[2]) = 26 bytes`

Let's see an example of an array of structures that stores information of 5 students and prints it.

```
1. #include<stdio.h>
2. #include <string.h>
3. struct student{
4. int rollno;
5. char name[10];
6. };
7. int main(){
8. int i;
9. struct student st[5];
10.printf("Enter Records of 5 students");
11.for(i=0;i<5;i++){
12.printf("\nEnter Rollno:");
13 scanf("%d",&st[i].rollno);
14.printf("\nEnter Name:");
15 scanf("%s",&st[i].name);
16.}
17.printf("\nStudent Information List:");
18.for(i=0;i<5;i++){
19.printf("\nRollno:%d, Name:%s",st[i].rollno,st[i].name);
20.}
21. return 0;
22.}
```

Output:

```
Enter Records of 5 students
Enter Rollno:1
Enter Name:Sonoo
Enter Rollno:2
```

```
Enter Name:Ratan
Enter Rollno:3
Enter Name:Vimal
Enter Rollno:4
Enter Name:James
Enter Rollno:5
Enter Name:Sarfraz
```

Student Information List:

```
Rollno:1, Name:Sonoo
Rollno:2, Name:Ratan
Rollno:3, Name:Vimal
Rollno:4, Name:James
```

```
Rollno:5, Name:Sarfraz
```

Nested Structure in C

C provides us the feature of nesting one structure within another structure by using which, complex data types are created. For example, we may need to store the address of an entity employee in a structure. The attribute address may also have the subparts as street number, city, state, and pin code. Hence, to store the address of the employee, we need to store the address of the employee into a separate structure and nest the structure address into the structure employee. Consider the following program.

1. `#include<stdio.h>`
2. `struct address`
3. `{`
4. `char city[20];`
5. `int pin;`
6. `char phone[14];`
7. `};`
8. `struct employee`
9. `{`

```
10. char name[20];
11. struct address add;
12.};
13.void main ()
14.{
15. struct employee emp;
16. printf("Enter employee information?\n");
17. scanf("%s %s %d %s",emp.name,emp.add.city, &emp.add.pin,
    emp.add.phone);
18. printf("Printing the employee information....\n");
19. printf("name: %s\nCity: %s\nPincode: %d\nPhone:
    %s",emp.name,emp.add.city,emp.add.pin,emp.add.phone);
20.}
```

Output

Enter employee information?

Arun

Delhi

110001

1234567890

Printing the employee information....

name: Arun

City: Delhi

Pincode: 110001

Phone: 1234567890

The structure can be nested in the following ways.

1. By separate structure
2. By Embedded structure

1) Separate structure

Here, we create two structures, but the dependent structure should be used inside the main structure as a member. Consider the following example.

1. `struct Date`
2. `{`
3. `int dd;`
4. `int mm;`
5. `int yyyy;`
6. `};`
7. `struct Employee`
8. `{`
9. `int id;`
10. `char name[20];`
11. `struct Date doj;`
12. `}emp1;`

As you can see, doj (date of joining) is the variable of type Date. Here doj is used as a member in Employee structure. In this way, we can use Date structure in many structures.

2) Embedded structure

The embedded structure enables us to declare the structure inside the structure. Hence, it requires less line of codes but it can not be used in multiple data structures. Consider the following example.

```
1. struct Employee
2. {
3.     int id;
4.     char name[20];
5.     struct Date
6.     {
7.         int dd;
8.         int mm;
9.         int yyyy;
10.    }doj;
11.}emp1;
```

Accessing Nested Structure

We can access the member of the nested structure by Outer_Structure.Nested_Structure.member as given below:

```
1. e1.doj.dd
2. e1.doj.mm
3. e1.doj.yyyy
```

C Nested Structure example

Let's see a simple example of the nested structure in C language.

```
1. #include <stdio.h>
2. #include <string.h>
3. struct Employee
4. {
5.     int id;
6.     char name[20];
```



```

7.  struct Date
8.  {
9.      int dd;
10.     int mm;
11.     int yyyy;
12. }doj;
13.}e1;
14.int main()
15.{
16.    //storing employee information
17.    e1.id=101;
18.    strcpy(e1.name, "Sonoo Jaiswal");//copying string into char array
19.    e1.doj.dd=10;
20.    e1.doj.mm=11;
21.    e1.doj.yyyy=2014;
22.
23.    //printing first employee information
24.    printf( "employee id : %d\n", e1.id);
25.    printf( "employee name : %s\n", e1.name);
26.    printf( "employee date of joining (dd/mm/yyyy) : %d/%d/%d\n",
            e1.doj.dd,e1.doj.mm,e1.doj.yyyy);
27.    return 0;
28.}

```

Output:

```

employee id : 101
employee name : Sonoo Jaiswal
employee date of joining (dd/mm/yyyy) : 10/11/2014

```

Passing structure to function

Just like other variables, a structure can also be passed to a function. We may pass the structure members into the function or pass the structure variable at once. Consider the following example to pass the structure variable employee to a function display() which is used to display the details of an employee.

```
1. #include<stdio.h>
2. struct address
3. {
4.     char city[20];
5.     int pin;
6.     char phone[14];
7. };
8. struct employee
9. {
10.    char name[20];
11.    struct address add;
12.};
13.void display(struct employee);
14.void main ()
15.{
16.    struct employee emp;
17.    printf("Enter employee information?\n");
18.    scanf("%s %s %d %s",emp.name,emp.add.city, &emp.add.pin,
        emp.add.phone);
19.    display(emp);
20.}
21.void display(struct employee emp)
22.{
```

```
23. printf("Printing the details...\n");
24. printf("%s %s %d %s",emp.name,emp.add.city,emp.add.pin,emp.add.phone);
25.}
```

Structure Padding in C

Structure padding is a concept in C that adds the one or more empty bytes between the memory addresses to align the data in memory.

Let's first understand the structure padding in C through a simple scenario which is given below:

Suppose we create a user-defined structure. When we create an object of this structure, then the contiguous memory will be allocated to the structure members.

```
1. struct student
2. {
3.     char a;
4.     char b;
5.     int c;
6. } stud1;
```

In the above example, we have created a structure of type student. We have declared the object of this structure named as 'stud1'. After the creation of an object, a contiguous block of memory is allocated to its structure members. First, the memory will be allocated to the 'a' variable, then 'b' variable, and then 'c' variable.

What is the size of the struct student?

Now, we calculate the size of the struct student. We assume that the size of the int is 4 bytes, and the size of the char is 1 byte.

1. **struct** student
2. {
3. **char** a; // 1 byte
4. **char** b; // 1 byte
5. **int** c; // 4 bytes
6. }

In the above case, when we calculate the size of the struct student, size comes to be 6 bytes. But this answer is wrong. Now, we will understand why this answer is wrong? We need to understand the concept of structure padding.

Structure Padding

The processor does not read 1 byte at a time. It reads 1 word at a time.

What does the 1 word mean?

If we have a 32-bit processor, then the processor reads 4 bytes at a time, which means that 1 word is equal to 4 bytes.

1. 1 word = 4 bytes

If we have a 64-bit processor, then the processor reads 8 bytes at a time, which means that 1 word is equal to 8 bytes.

1. 1 word = 8 bytes

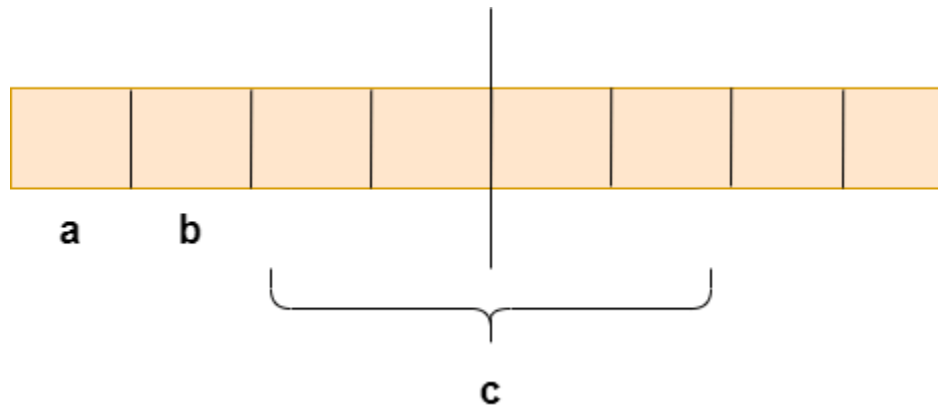
Therefore, we can say that a 32-bit processor is capable of accessing 4 bytes at a time, whereas a 64-bit processor is capable of accessing 8 bytes at a time. It depends upon the architecture that what would be the size of the word.

Why structure padding?

1. **struct** student
2. {
3. **char** a; // 1 byte
4. **char** b; // 1 byte

5. `int c; // 4 bytes`
6. `}`

If we have a 32-bit processor (4 bytes at a time), then the pictorial representation of the memory for the above structure would be:



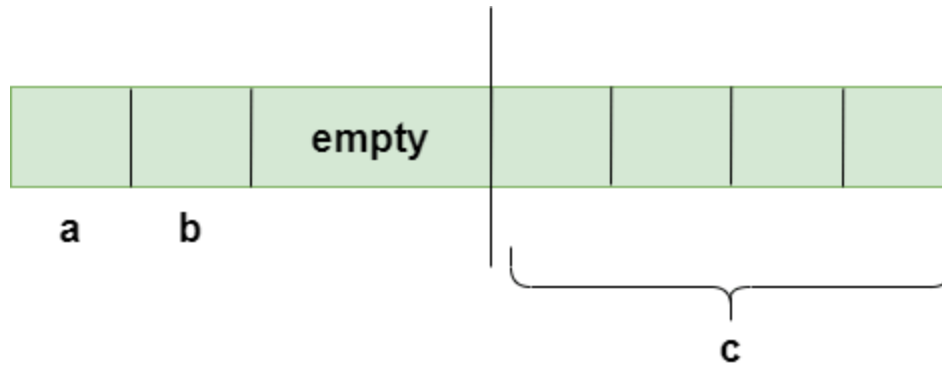
As we know that structure occupies the contiguous block of memory as shown in the above diagram, i.e., 1 byte for char a, 1 byte for char b, and 4 bytes for int c, then what problem do we face in this case.

What's the problem?

The 4-bytes can be accessed at a time as we are considering the 32-bit architecture. The problem is that in one CPU cycle, one byte of char a, one byte of char b, and 2 bytes of int c can be accessed. We will not face any problem while accessing the char a and char b as both the variables can be accessed in one CPU cycle, but we will face the problem when we access the int c variable as 2 CPU cycles are required to access the value of the 'c' variable. In the first CPU cycle, the first two bytes are accessed, and in the second cycle, the other two bytes are accessed.

Suppose we do not want to access the 'a' and 'b' variable, we only want to access the variable 'c', which requires two cycles. The variable 'c' is of 4 bytes, so it can be accessed in one cycle also, but in this scenario, it is utilizing 2 cycles. This is an unnecessary wastage of CPU cycles. Due to this reason, the structure padding concept was introduced to save the number of CPU cycles. The structure padding is done automatically by the compiler. Now, we will see how structure padding is done.

How is structure padding done?



In order to achieve the structure padding, an empty row is created on the left, as shown in the above diagram, and the two bytes which are occupied by the 'c' variable on the left are shifted to the right. So, all the four bytes of 'c' variable are on the right. Now, the 'c' variable can be accessed in a single CPU cycle. After structure padding, the total memory occupied by the structure is 8 bytes (1 byte+1 byte+2 bytes+4 bytes), which is greater than the previous one. Although the memory is wasted in this case, the variable can be accessed within a single cycle.

Let's create a simple program of structures.

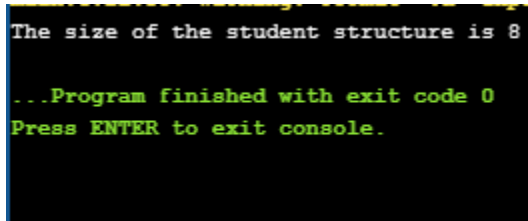
```

1. #include <stdio.h>
2. struct student
3. {
4.     char a;
5.     char b;
6.     int c;
7. };
8. int main()
9. {
10.    struct student stud1; // variable declaration of the student type..
11.    // Displaying the size of the structure student.
12.    printf("The size of the student structure is %d", sizeof(stud1));
13.    return 0;
14.}

```

In the above code, we have created a structure named 'student'. Inside the main() method, we declare a variable of student type, i.e., stud1, and then we calculate the size of the student by using the sizeof() operator. The output would be 8 bytes due to the concept of structure padding, which we have already discussed in the above.

Output



```
The size of the student structure is 8

...Program finished with exit code 0
Press ENTER to exit console.
```

Changing order of the variables

Now, we will see what happens when we change the order of the variables, does it affect the output of the program. Let's consider the same program.

1. `#include <stdio.h>`
2. `struct student`
3. `{`
4. `char a;`
5. `int b;`
6. `char c;`
7. `};`
8. `int main()`
9. `{`
10. `struct student stud1; // variable declaration of the student type..`
11. `// Displaying the size of the structure student.`
12. `printf("The size of the student structure is %d", sizeof(stud1));`
13. `return 0;`
14. `}`

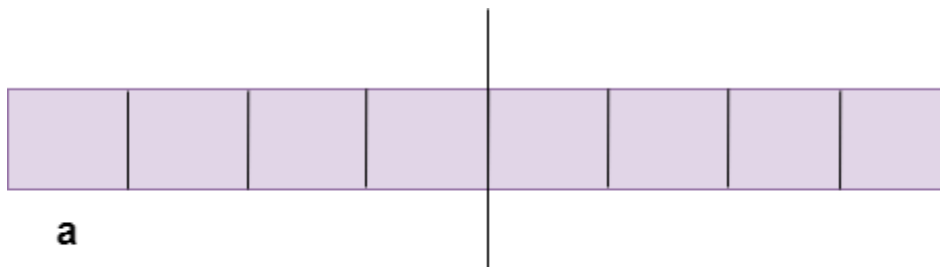
The above code is similar to the previous code; the only thing we change is the order of the variables inside the structure student. Due to the change in the order, the output would be different in both the cases. In the previous case, the output was 8 bytes, but in this case, the output is 12 bytes, as we can observe in the below screenshot.

Output

```
The size of the student structure is 12
...Program finished with exit code 0
Press ENTER to exit console.□
```

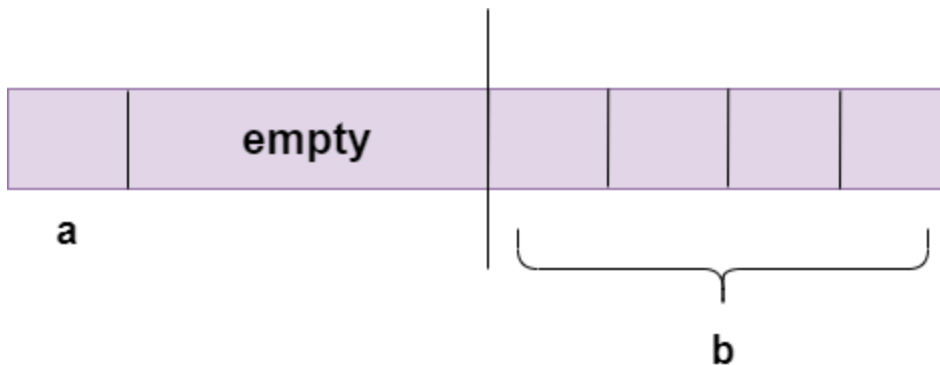
Now, we need to understand "why the output is different in this case".

- First, memory is allocated to the char a variable, i.e., 1 byte.

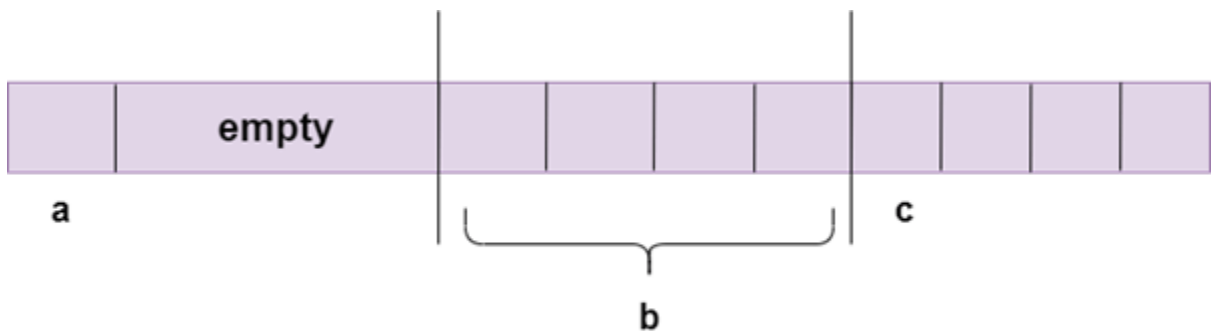


- Now, the memory will be allocated to the int b. Since the int variable occupies 4 bytes, but on the left, only 3 bytes are available. The empty row will be created on these 3 bytes, and the int variable would occupy the other 4 bytes so that the

integer variable can be accessed in a single CPU cycle.



- Now, the memory will be given to the char c At a time, CPU can access 1 word, which is equal to 4 bytes, so CPU will use 4 bytes to access a 'c' variable. Therefore, the total memory required is 12 bytes (4 bytes +4 bytes +4 bytes), i.e., 4 bytes required to access char a variable, 4 bytes required to access int b variable, and other 4 bytes required to access a single character of 'c' variable.



How to avoid the structure padding in C?

The structural padding is an in-built process that is automatically done by the compiler. Sometimes it required to avoid the structure padding in C as it makes the size of the structure greater than the size of the structure members.

We can avoid the structure padding in C in two ways:

- Using `#pragma pack(1)` directive

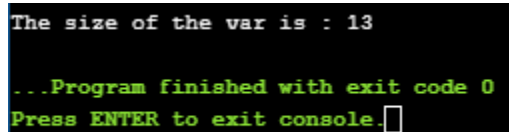
- Using attribute

Using #pragma pack(1) directive

```
1. #include <stdio.h>
2. #pragma pack(1)
3. struct base
4. {
5.     int a;
6.     char b;
7.     double c;
8. };
9. int main()
10.{
11. struct base var; // variable declaration of type base
12. // Displaying the size of the structure base
13. printf("The size of the var is : %d", sizeof(var));
14.return 0;
15.}
```

In the above code, we have used the #pragma pack(1) directive to avoid the structure padding. If we do not use this directive, then the output of the above program would be 16 bytes. But the actual size of the structure members is 13 bytes, so 3 bytes are wasted. To avoid the wastage of memory, we use the #pragma pack(1) directive to provide the 1-byte packaging.

Output



```
The size of the var is : 13
...Program finished with exit code 0
Press ENTER to exit console.
```

- By using attribute

```
1. #include <stdio.h>
2.
3. struct base
4. {
5.     int a;
6.     char b;
7.     double c;
8. }__attribute__((packed)); ;
9. int main()
10.{
11. struct base var; // variable declaration of type base
12. // Displaying the size of the structure base
13. printf("The size of the var is : %d", sizeof(var));
14.
15. return 0;
16.}
```

Union in C

Union can be defined as a user-defined data type which is a collection of different variables of different data types in the same memory location. The union can also be defined as many members, but only one member can contain a value at a particular point in time.

Union is a user-defined data type, but unlike structures, they share the same memory location.

Let's understand this through an example.

```
1. struct abc
2. {
3.     int a;
4.     char b;
5. }
```

The above code is the user-defined structure that consists of two members, i.e., 'a' of type int and 'b' of type character. When we check the addresses of 'a' and 'b', we found that their addresses are different. Therefore, we conclude that the members in the structure do not share the same memory location.

When we define the union, then we found that union is defined in the same way as the structure is defined but the difference is that union keyword is used for defining the union data type, whereas the struct keyword is used for defining the structure. The union contains the data members, i.e., 'a' and 'b', when we check the addresses of both the variables then we found that both have the same addresses. It means that the union members share the same memory location.

Let's have a look at the pictorial representation of the memory allocation.

The below figure shows the pictorial representation of the structure. The structure has two members; i.e., one is of integer type, and the another one is of character type. Since 1 block is equal to 1 byte; therefore, 'a' variable will be allocated 4 blocks of memory while 'b' variable will be allocated 1 block of memory.

The below figure shows the pictorial representation of union members. Both the variables are sharing the same memory location and having the same initial address.

In union, members will share the memory location. If we try to make changes in any of the member then it will be reflected to the other member as well. Let's understand this concept through an example.

```
1. union abc
2. {
3.     int a;
4.     char b;
5. }var;
6. int main()
7. {
8.     var.a = 66;
9.     printf("\n a = %d", var.a);
10.    printf("\n b = %d", var.b);
11.}
```

In the above code, union has two members, i.e., 'a' and 'b'. The 'var' is a variable of union abc type. In the main() method, we assign the 66 to 'a' variable, so var.a will print 66 on the screen. Since both 'a' and 'b' share the memory location, var.b will print 'B' (ascii code of 66).

Deciding the size of the union

The size of the union is based on the size of the largest member of the union.

Let's understand through an example.

```
1. union abc{
2.   int a;
3.   char b;
4.   float c;
5.   double d;
6. };
7. int main()
8. {
9.   printf("Size of union abc is %d", sizeof(union abc));
10. return 0;
11.}
```

As we know, the size of int is 4 bytes, size of char is 1 byte, size of float is 4 bytes, and the size of double is 8 bytes. Since the double variable occupies the largest memory among all the four variables, so total 8 bytes will be allocated in the memory. Therefore, the output of the above program would be 8 bytes.

Accessing members of union using pointers

We can access the members of the union through pointers by using the (->) arrow operator.

Let's understand through an example.

```
1. #include <stdio.h>
2. union abc
```

```
3. {
4.     int a;
5.     char b;
6. };
7. int main()
8. {
9.     union abc *ptr; // pointer variable declaration
10.    union abc var;
11.    var.a= 90;
12.    ptr = &var;
13.    printf("The value of a is : %d", ptr->a);
14.    return 0;
15.}
```

In the above code, we have created a pointer variable, i.e., *ptr, that stores the address of var variable. Now, ptr can access the variable 'a' by using the (->) operator. Hence the output of the above code would be 90.

Why do we need C unions?

Consider one example to understand the need for C unions. Let's consider a store that has two items:

- Books
- Shirts

Store owners want to store the records of the above-mentioned two items along with the relevant information. For example, Books include Title, Author, no of pages, price, and Shirts include Color, design, size, and price. The 'price' property is common in both

items. The Store owner wants to store the properties, then how he/she will store the records.

Initially, they decided to store the records in a structure as shown below:

```
1. struct store
2. {
3.     double price;
4.     char *title;
5.     char *author;
6.     int number_pages;
7.     int color;
8.     int size;
9.     char *design;
10.};
```

The above structure consists of all the items that store owner wants to store. The above structure is completely usable but the price is common property in both the items and the rest of the items are individual. The properties like price, *title, *author, and number_pages belong to Books while color, size, *design belongs to Shirt.

Let's see how can we access the members of the structure.

```
1. int main()
2. {
3.     struct store book;
4.     book.title = "C programming";
5.     book.author = "Paulo Cohelo";
6.     book.number_pages = 190;
7.     book.price = 205;
8.     printf("Size is : %ld bytes", sizeof(book));
```



```
9. return 0;
10.}
```

In the above code, we have created a variable of type store. We have assigned the values to the variables, title, author, number_pages, price but the book variable does not possess the properties such as size, color, and design. Hence, it's a wastage of memory. The size of the above structure would be 44 bytes.

We can save lots of space if we use unions.

```
1. #include <stdio.h>
2. struct store
3. {
4.     double price;
5.     union
6.     {
7.         struct{
8.             char *title;
9.             char *author;
10.            int number_pages;
11.        } book;
12.
13.        struct {
14.            int color;
15.            int size;
16.            char *design;
17.        } shirt;
18.    }item;
19.};
20. int main()
```

```
21.{  
22.  struct store s;  
23.  s.item.book.title = "C programming";  
24.  s.item.book.author = "John";  
25.  s.item.book.number_pages = 189;  
26.  printf("Size is %ld", sizeof(s));  
27.  return 0;  
28.}
```

In the above code, we have created a variable of type store. Since we used the unions in the above code, so the largest memory occupied by the variable would be considered for the memory allocation. The output of the above program is 32 bytes. In the case of structures, we obtained 44 bytes, while in the case of unions, the size obtained is 44 bytes. Hence, 44 bytes is greater than 32 bytes saving lots of memory space.

File Handling in C

In programming, we may require some specific input data to be generated several numbers of times. Sometimes, it is not enough to only display the data on the console. The data to be displayed may be very large, and only a limited amount of data can be displayed on the console, and since the memory is volatile, it is impossible to recover the programmatically generated data again and again. However, if we need to do so, we may store it onto the local file system which is volatile and can be accessed every time. Here, comes the need of file handling in C.

File handling in C enables us to create, update, read, and delete the files stored on the local file system through our C program. The following operations can be performed on a file.

- **Creation of the new file**
 - **Opening an existing file**
 - **Reading from the file**
 - **Writing to the file**
 - **Deleting the file**
-

Functions for file handling

There are many functions in the C library to open, read, write, search and close the file.
A list of file functions are given below:

No.	Function	Description
1	fopen()	opens new or existing file
2	fprintf()	write data into the file
3	fscanf()	reads data from the file
4	fputc()	writes a character into the file

5	fgetc()	reads a character from file
6	fclose()	closes the file
7	fseek()	sets the file pointer to given position
8	fputw()	writes an integer to file
9	fgetw()	reads an integer from file
10	ftell()	returns current position
11	rewind()	sets the file pointer to the beginning of the file

Opening File: fopen()

We must open a file before it can be read, write, or update. The `fopen()` function is used to open a file. The syntax of the `fopen()` is given below.

1. `FILE *fopen(const char * filename, const char * mode);`

The `fopen()` function accepts two parameters:

- The file name (string). If the file is stored at some specific location, then we must mention the path at which the file is stored. For example, a file name can be like "c://some_folder/some_file.ext".
- The mode in which the file is to be opened. It is a string.

We can use one of the following modes in the `fopen()` function.

Mode	Description
r	opens a text file in read mode
w	opens a text file in write mode
a	opens a text file in append mode
r+	opens a text file in read and write mode
w+	opens a text file in read and write mode
a+	opens a text file in read and write mode
rb	opens a binary file in read mode

wb	opens a binary file in write mode
ab	opens a binary file in append mode
rb+	opens a binary file in read and write mode
wb+	opens a binary file in read and write mode
ab+	opens a binary file in read and write mode

The fopen function works in the following way.

- Firstly, It searches the file to be opened.
- Then, it loads the file from the disk and place it into the buffer. The buffer is used to provide efficiency for the read operations.
- It sets up a character pointer which points to the first character of the file.

Consider the following example which opens a file in write mode.

```
1. #include<stdio.h>
2. void main( )
3. {
4. FILE *fp ;
5. char ch ;
6. fp = fopen("file_handle.c","r");
7. while ( 1 )
8. {
9. ch = fgetc ( fp ) ;
10.if ( ch == EOF )
11.break ;
12.printf("%c",ch) ;
13.}
14.fclose (fp ) ;
15.}
```

Output

The content of the file will be printed.

```
#include;

void main( )

{

FILE *fp; // file pointer

char ch;

fp = fopen("file_handle.c","r");

while ( 1 )
```

```
(  
  
ch = fgetc ( fp ); //Each character of the file is read and stored in the character file.  
  
if ( ch == EOF )  
  
break;  
  
printf("%c",ch);  
  
)  
  
fclose (fp );  
  
)
```

Closing File: fclose()

The `fclose()` function is used to close a file. The file must be closed after performing all the operations on it. The syntax of `fclose()` function is given below:

1. `int fclose(FILE *fp);`

C fprintf() and fscanf()

Writing File : fprintf() function

The `fprintf()` function is used to write set of characters into file. It sends formatted output to a stream.

Syntax:

1. `int fprintf(FILE *stream, const char *format [, argument, ...])`

Example:

1. `#include <stdio.h>`
 2. `main(){`
 3. `FILE *fp;`
 4. `fp = fopen("C:\Users\Anil Kanaji\Desktop\Testing\JUnit\file.txt",`
`"w");//opening file`
 5. `fprintf(fp, "Hello file by fprintf...\n");//writing data into file`
 6. `fclose(fp);//closing file`
 7. `}`
-

Reading File : fscanf() function

The `fscanf()` function is used to read set of characters from file. It reads a word from the file and returns EOF at the end of file.

Syntax:

1. `int fscanf(FILE *stream, const char *format [, argument, ...])`

Example:

1. `#include <stdio.h>`
2. `main(){`
3. `FILE *fp;`

```
4.  char buff[255]; //creating char array to store data of file
5.  fp = fopen("file.txt", "r");
6.  while(fscanf(fp, "%s", buff) != EOF){
7.      printf("%s ", buff);
8.  }
9.  fclose(fp);
10. }
```

Output:

Hello file by fprintf...



C File Example: Storing employee information

Let's see a file handling example to store employee information as entered by user from console. We are going to store id, name and salary of the employee.

```
1.  #include <stdio.h>
2.  void main()
3.  {
4.      FILE *fptr;
5.      int id;
6.      char name[30];
7.      float salary;
8.      fptr = fopen("emp.txt", "w+"); /* open for writing */
9.      if (fptr == NULL)
10.     {
```

```
11.    printf("File does not exists \n");
12.    return;
13. }
14.    printf("Enter the id\n");
15.    scanf("%d", &id);
16.    fprintf(fp, "Id= %d\n", id);
17.    printf("Enter the name \n");
18.    scanf("%s", name);
19.    fprintf(fp, "Name= %s\n", name);
20.    printf("Enter the salary\n");
21.    scanf("%f", &salary);
22.    fprintf(fp, "Salary= %.2f\n", salary);
23.    fclose(fp);
24.}
```

Output:

Enter the id

1

Enter the name

sonoo

Enter the salary

120000



Now open file from current directory. For windows operating system, go to TC\bin directory, you will see emp.txt file. It will have following information.

emp.txt

Id= 1

Name= sonoo

Salary= 120000

C fputc() and fgetc()

Writing File : fputc() function

The fputc() function is used to write a single character into file. It outputs a character to a stream.

Syntax:

1. **int fputc(int c, FILE *stream)**

Example:

1. **#include <stdio.h>**
2. **main(){**
3. **FILE *fp;**
4. **fp = fopen("file1.txt", "w");//opening file**
5. **fputc('a',fp);//writing single character into file**
6. **fclose(fp);//closing file**
7. **}**

file1.txt

Reading File : fgetc() function

The fgetc() function returns a single character from the file. It gets a character from the stream. It returns EOF at the end of file.

Syntax:

1. `int fgetc(FILE *stream)`

Example:

1. `#include<stdio.h>`
2. `#include<conio.h>`
3. `void main(){`
4. `FILE *fp;`
5. `char c;`
6. `clrscr();`
7. `fp=fopen("myfile.txt","r");`
- 8.
9. `while((c=fgetc(fp))!=EOF){`
10. `printf("%c",c);`
11. `}`
12. `fclose(fp);`
13. `getch();`
14. `}`

myfile.txt

this is simple text message

C fputs() and fgets()

The fputs() and fgets() in C programming are used to write and read string from stream. Let's see examples of writing and reading file using fgets() and fputs() functions.

Writing File : fputs() function

The fputs() function writes a line of characters into file. It outputs string to a stream.

Syntax:

1. **int** fputs(**const char** *s, **FILE** *stream)

Example:

1. **#include**<stdio.h>
2. **#include**<conio.h>
3. **void** main(){
4. **FILE** *fp;
5. clrscr();
- 6.
7. fp=fopen("myfile2.txt","w");
8. fputs("hello c programming",fp);
- 9.
10. fclose(fp);
11. getch();

12.}

myfile2.txt

hello c programming

Reading File : fgets() function

The fgets() function reads a line of characters from file. It gets string from a stream.

Syntax:

1. **char*** fgets(**char** *s, **int** n, **FILE** *stream)

Example:

```
1. #include<stdio.h>
2. #include<conio.h>
3. void main(){
4. FILE *fp;
5. char text[300];
6. clrscr();
7.
8. fp=fopen("myfile2.txt","r");
9. printf("%s",fgets(text,200,fp));
10.
11. fclose(fp);
12. getch();
13.}
```

Output:

hello c programming

C fseek() function

The `fseek()` function is used to set the file pointer to the specified offset. It is used to write data into file at desired location.

Syntax:

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

There are 3 constants used in the `fseek()` function for whence: `SEEK_SET`, `SEEK_CUR` and `SEEK_END`.

Example:

1. `#include <stdio.h>`
2. `void main(){`
3. `FILE *fp;`
- 4.
5. `fp = fopen("myfile.txt","w+");`
6. `fputs("This is javatpoint", fp);`
- 7.
8. `fseek(fp, 7, SEEK_SET);`
9. `fputs("sonoo jaiswal", fp);`
10. `fclose(fp);`
11. `}`

myfile.txt

This is sonoo jaiswa

C rewind() function

The `rewind()` function sets the file pointer at the beginning of the stream. It is useful if you have to use stream many times.

Syntax:

1. `void rewind(FILE *stream)`

Example:

File: file.txt

1. this is a simple text

File: rewind.c

```
1. #include<stdio.h>
2. #include<conio.h>
3. void main(){
4. FILE *fp;
5. char c;
6. clrscr();
7. fp=fopen("file.txt","r");
8.
9. while((c=fgetc(fp))!=EOF){
10.printf("%c",c);
11.}
12.
13.rewind(fp);//moves the file pointer at beginning of the file
14.
15.while((c=fgetc(fp))!=EOF){
16.printf("%c",c);
17.}
18.
19 fclose(fp);
20.getch();
21.}
```

Output:

this is a simple textthis is a simple text

As you can see, rewind() function moves the file pointer at beginning of the file that is why "this is simple text" is printed 2 times. If you don't call rewind() function, "this is simple text" will be printed only once.

C ftell() function

The `ftell()` function returns the current file position of the specified stream. We can use `ftell()` function to get the total size of a file after moving file pointer at the end of file. We can use `SEEK_END` constant to move the file pointer at the end of file.

Syntax:

1. `long int ftell(FILE *stream)`

Example:

File: ftell.c

1. `#include <stdio.h>`
2. `#include <conio.h>`
3. `void main ()`
4. `{`
5. `FILE *fp;`
6. `int length;`
7. `clrscr();`
8. `fp = fopen("file.txt", "r");`
9. `fseek(fp, 0, SEEK_END);`
10. `length = ftell(fp);`
11.
12. `fclose(fp);`
13. `printf("Size of file: %d bytes", length);`
14. `getch();`
15. `}`

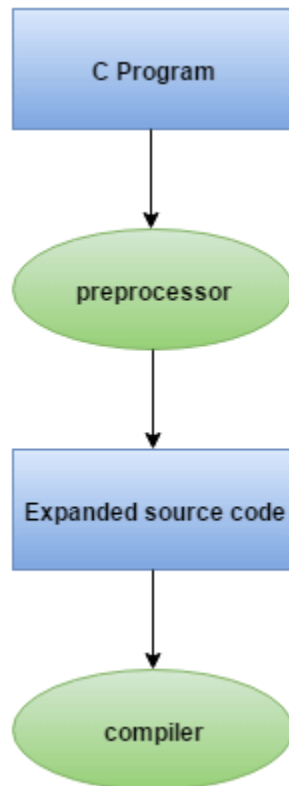
Output:

Size of file: 21 bytes

C Preprocessor Directives

The C preprocessor is a micro processor that is used by compiler to transform your code before compilation. It is called micro preprocessor because it allows us to add macros.

Note: Preprocessor directives are executed before compilation.



All preprocessor directives starts with hash # symbol.

Let's see a list of preprocessor directives.

- **#include**
- **#define**
- **#undef**
- **#ifdef**
- **#ifndef**
- **#if**
- **#else**
- **#elif**
- **#endif**
- **#error**
- **#pragma**

C Macros

A macro is a segment of code which is replaced by the value of macro. Macro is defined by #define directive. There are two types of macros:

1. Object-like Macros
2. Function-like Macros

Object-like Macros

The object-like macro is an identifier that is replaced by value. It is widely used to represent numeric constants. For example:

1. `#define PI 3.14`

Here, PI is the macro name which will be replaced by the value 3.14.

Function-like Macros

The function-like macro looks like function call. For example:

1. `#define MIN(a,b) ((a)<(b)?(a):(b))`

Here, MIN is the macro name.

Visit [#define](#) to see the full example of object-like and function-like macros.

C Predefined Macros

ANSI C defines many predefined macros that can be used in c program.

No.	Macro	Description
1	<code>_DATE_</code>	represents current date in "MMM DD YYYY" format.
2	<code>_TIME_</code>	represents current time in "HH:MM:SS" format.
3	<code>_FILE_</code>	represents current file name.

4	<code>__LINE__</code>	represents current line number.
5	<code>__STDC__</code>	It is defined as 1 when compiler complies with the ANSI standard.

C predefined macros example

File: *simple.c*

```

1. #include<stdio.h>
2. int main(){
3.     printf("File :%s\n", __FILE__ );
4.     printf("Date :%s\n", __DATE__ );
5.     printf("Time :%s\n", __TIME__ );
6.     printf("Line :%d\n", __LINE__ );
7.     printf("STDC :%d\n", __STDC__ );
8.     return 0;
9. }
```

Output:

File :simple.c

Date :Dec 6 2015

Time :12:28:46

Line :6

STDC :1

C #include

The `#include` preprocessor directive is used to paste code of given file into current file. It is used include system-defined and user-defined header files. If included file is not found, compiler renders error.

By the use of `#include` directive, we provide information to the preprocessor where to look for the header files. There are two variants to use `#include` directive.

1. `#include <filename>`
2. `#include "filename"`

The `#include <filename>` tells the compiler to look for the directory where system header files are held. In UNIX, it is `\usr\include` directory.

The `#include "filename"` tells the compiler to look in the current directory from where program is running.

#include directive example

Let's see a simple example of `#include` directive. In this program, we are including `stdio.h` file because `printf()` function is defined in this file.

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

Output:

Hello C



#include notes:

Note 1: In #include directive, comments are not recognized. So in case of #include <a//b>, a//b is treated as filename.

Note 2: In #include directive, backslash is considered as normal text not escape sequence. So in case of #include <a\nb>, a\nb is treated as filename.

Note 3: You can use only comment after filename otherwise it will give error.

C #define

The #define preprocessor directive is used to define constant or micro substitution. It can use any basic data type.

Syntax:

1. **#define token value**

Let's see an example of #define to define a constant.

1. **#include <stdio.h>**
2. **#define PI 3.14**
3. **main() {**
4. **printf("%f",PI);**
5. **}**

Output:

3.140000



Let's see an example of #define to create a macro.

1. `#include <stdio.h>`
2. `#define MIN(a,b) ((a)<(b)?(a):(b))`
3. `void main() {`
4. `printf("Minimum between 10 and 20 is: %d\n", MIN(10,20));`
5. `}`

Output:

Minimum between 10 and 20 is: 10

C #undef

The `#undef` preprocessor directive is used to undefine the constant or macro defined by `#define`.

Syntax:

1. `#undef token`

Let's see a simple example to define and undefine a constant.

1. `#include <stdio.h>`
2. `#define PI 3.14`
3. `#undef PI`
4. `main() {`
5. `printf("%f",PI);`
6. `}`

Output:

Compile Time Error: 'PI' undeclared

The `#undef` directive is used to define the preprocessor constant to a limited scope so that you can declare constant again.

Let's see an example where we are defining and undefining number variable. But before being undefined, it was used by square variable.

1. `#include <stdio.h>`
2. `#define number 15`
3. `int square=number*number;`
4. `#undef number`
5. `main() {`
6. `printf("%d",square);`
7. `}`

Output:

225

C #ifdef

The `#ifdef` preprocessor directive checks if macro is defined by `#define`. If yes, it executes the code otherwise `#else` code is executed, if present.

Syntax:

1. `#ifdef MACRO`
2. `//code`
3. `#endif`

Syntax with `#else`:

1. `#ifdef MACRO`

2. `//successful code`
3. `#else`
4. `//else code`
5. `#endif`

C #ifdef example

Let's see a simple example to use #ifdef preprocessor directive.

1. `#include <stdio.h>`
2. `#include <conio.h>`
3. `#define NOINPUT`
4. `void main() {`
5. `int a=0;`
6. `#ifdef NOINPUT`
7. `a=2;`
8. `#else`
9. `printf("Enter a:");`
10. `scanf("%d", &a);`
11. `#endif`
12. `printf("Value of a: %d\n", a);`
13. `getch();`
14. `}`

Output:

Value of a: 2



But, if you don't define NOINPUT, it will ask user to enter a number.

1. `#include <stdio.h>`
2. `#include <conio.h>`
3. `void main() {`
4. `int a=0;`
5. `#ifdef NOINPUT`
6. `a=2;`
7. `#else`
8. `printf("Enter a:");`
9. `scanf("%d", &a);`
10. `#endif`
11.
12. `printf("Value of a: %d\n", a);`
13. `getch();`
14. `}`

Output:

Enter a:5

Value of a: 5

C #ifndef

The `#ifndef` preprocessor directive checks if macro is not defined by `#define`. If yes, it executes the code otherwise `#else` code is executed, if present.

Syntax:

1. `#ifndef MACRO`
2. `//code`
3. `#endif`

Syntax with #else:

1. `#ifndef MACRO`
2. `//successful code`
3. `#else`
4. `//else code`
5. `#endif`

C #ifndef example

Let's see a simple example to use #ifndef preprocessor directive.

1. `#include <stdio.h>`
2. `#include <conio.h>`
3. `#define INPUT`
4. `void main() {`
5. `int a=0;`
6. `#ifndef INPUT`
7. `a=2;`
8. `#else`
9. `printf("Enter a:");`
10. `scanf("%d", &a);`
11. `#endif`
12. `printf("Value of a: %d\n", a);`
13. `getch();`
14. `}`

Output:

Enter a:5

Value of a: 5

But, if you don't define INPUT, it will execute the code of #ifndef.

```
1. #include <stdio.h>
2. #include <conio.h>
3. void main() {
4.     int a=0;
5.     #ifndef INPUT
6.     a=2;
7.     #else
8.     printf("Enter a:");
9.     scanf("%d", &a);
10. #endif
11. printf("Value of a: %d\n", a);
12. getch();
13. }
```

Output:

Value of a: 2

C #if

The #if preprocessor directive evaluates the expression or condition. If condition is true, it executes the code otherwise #elseif or #else or #endif code is executed.

Syntax:

```
1. #if expression
2. //code
```

3. `#endif`

Syntax with `#else`:

1. `#if expression`
2. `//if code`
3. `#else`
4. `//else code`
5. `#endif`

Syntax with `#elif` and `#else`:

1. `#if expression`
2. `//if code`
3. `#elif expression`
4. `//elif code`
5. `#else`
6. `//else code`
7. `#endif`

C `#if` example

Let's see a simple example to use `#if` preprocessor directive.

1. `#include <stdio.h>`
2. `#include <conio.h>`
3. `#define NUMBER 0`
4. `void main() {`
5. `#if (NUMBER==0)`
6. `printf("Value of Number is: %d",NUMBER);`
7. `#endif`


```
8. getch();
9. }
```

Output:

Value of Number is: 0



Let's see another example to understand the #if directive clearly.

```
1. #include <stdio.h>
2. #include <conio.h>
3. #define NUMBER 1
4. void main() {
5. clrscr();
6. #if (NUMBER==0)
7. printf("1 Value of Number is: %d",NUMBER);
8. #endif
9.
10.#if (NUMBER==1)
11.printf("2 Value of Number is: %d",NUMBER);
12.#endif
13.getch();
14.}
```

Output:

2 Value of Number is: 1



C #else

The `#else` preprocessor directive evaluates the expression or condition if condition of `#if` is false. It can be used with `#if`, `#elif`, `#ifdef` and `#ifndef` directives.

Syntax:

1. `#if expression`
2. `//if code`
3. `#else`
4. `//else code`
5. `#endif`

Syntax with `#elif`:

1. `#if expression`
2. `//if code`
3. `#elif expression`
4. `//elif code`
5. `#else`
6. `//else code`
7. `#endif`

C `#else` example

Let's see a simple example to use `#else` preprocessor directive.

1. `#include <stdio.h>`
2. `#include <conio.h>`
3. `#define NUMBER 1`
4. `void main() {`
5. `#if NUMBER==0`
6. `printf("Value of Number is: %d",NUMBER);`

```
7. #else
8. print("Value of Number is non-zero");
9. #endif
10.getch();
11.}
```

Output:

Value of Number is non-zero

C #error

The #error preprocessor directive indicates error. The compiler gives fatal error if #error directive is found and skips further compilation process.

C #error example

Let's see a simple example to use #error preprocessor directive.

```
1. #include<stdio.h>
2. #ifndef __MATH_H
3. #error First include then compile
4. #else
5. void main(){
6.     float a;
7.     a=sqrt(7);
8.     printf("%f",a);
9. }
10.#endif
```

Output:

Compile Time Error: First include then compile

But, if you include math.h, it does not gives error.

1. `#include<stdio.h>`
2. `#include<math.h>`
3. `#ifndef __MATH_H`
4. `#error First include then compile`
5. `#else`
6. `void main(){`
7. `float a;`
8. `a=sqrt(7);`
9. `printf("%f",a);`
10. `}`
11. `#endif`

Output:

2.645751

C #pragma

The `#pragma` preprocessor directive is used to provide additional information to the compiler. The `#pragma` directive is used by the compiler to offer machine or operating-system feature.

Syntax:

1. `#pragma token`

Different compilers can provide different usage of `#pragma` directive.

The turbo C++ compiler supports following #pragma directives.

1. `#pragma argsused`
2. `#pragma exit`
3. `#pragma hdrfile`
4. `#pragma hdrstop`
5. `#pragma inline`
6. `#pragma option`
7. `#pragma saveregs`
8. `#pragma startup`
9. `#pragma warn`

Let's see a simple example to use #pragma preprocessor directive.

1. `#include<stdio.h>`
2. `#include<conio.h>`
- 3.
4. `void func();`
- 5.
6. `#pragma startup func`
7. `#pragma exit func`
- 8.
9. `void main(){`
10. `printf("\nI am in main");`
11. `getch();`
12. `}`
- 13.
14. `void func(){`
15. `printf("\nI am in func");`
16. `getch();`

```
17.}
```

Output:

```
I am in func
```

```
I am in main
```

```
I am in func
```

Command Line Arguments in C

The arguments passed from command line are called command line arguments. These arguments are handled by `main()` function.

To support command line argument, you need to change the structure of `main()` function as given below.

```
1. int main(int argc, char *argv[] )
```

Here, `argc` counts the number of arguments. It counts the file name as the first argument.

The `argv[]` contains the total number of arguments. The first argument is the file name always.

Example

Let's see the example of command line arguments where we are passing one argument with file name.

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

```
2. void main(int argc, char *argv[] ) {
```

```
3.  
4.  printf("Program name is: %s\n", argv[0]);  
5.  
6.  if(argc < 2){  
7.      printf("No argument passed through command line.\n");  
8.  }  
9.  else{  
10.     printf("First argument is: %s\n", argv[1]);  
11. }  
12.}
```

Run this program as follows in Linux:

```
1. ./program hello
```

Run this program as follows in Windows from command line:

```
1. program.exe hello
```

Output:

Program name is: program

First argument is: hello



If you pass many arguments, it will print only one.

```
1. ./program hello c how r u
```

Output:

Program name is: program

First argument is: hello



But if you pass many arguments within double quote, all arguments will be treated as a single argument only.

1. `./program "hello c how r u"`

Output:

Program name is: program

First argument is: hello c how r u



You can write your program to print all the arguments. In this program, we are printing only `argv[1]`, that is why it is printing only one argument.