1-Introduction and coding of C language

1.1 Introduction of C language , Important and concept of Variables

What is C Language?

C is a general-purpose computer programming language developed in 1972 by Dennis Ritchie at the Bell Telephone Laboratories. As it was known as '**C**' because most of its features were taken from an earlier language called 'B'. C is highly portable and is used for scripting system applications which form a major part of Windows, UNIX, and Linux operating systems. It can efficiently work on enterprise applications, games, graphics, and applications requiring calculations, etc.

Candidates who love to create their own application or software by coding can learn this course easily and it is very simple to learn and understand. The basic knowledge of programming can be gained only from a **C Programming Language.** So, students get C language course certification from the popular institute and lead a programmer life in top MNCs, or International companies. Also, there are huge career opportunities for the C certified holder. Know the complete details of C Programming language

Concept of C language:

What is C?

The C programming language is a procedural and general-purpose language that provides low-level access to system memory. A program written in C must be run through a C compiler to convert it into an executable that a computer can run. Many versions of Unix-based operating systems (OSes) are written in C and it has been standardized as part of the Portable Operating System Interface (POSIX).

Today, the C programming language runs on many different hardware platforms and OSes such as Microsoft and Linux.

Pros and cons of C

The C language comes with a set of special characteristics, making it one of the most widely used languages of all time. The following are the main benefits of using C:

- **Structured.** It offers a structured programming approach for breaking down problems into smaller modules or functions that are easy to understand and modify.
- **Portable.** C is machine-independent and C programs can be executed on different machines.

- **Mid-level programming language.** It's a mid-level language that supports the features of both a low-level and a high-level language.
- **Rich library.** It offers numerous built-in library functions that expedite the development process.
- **Dynamic memory allocation.** C supports the dynamic memory allocation feature, which can be used to free the allocated memory at any time by calling the **free()** function.
- **Speed.** It's a compiler-based language, which makes the compilation and execution of code faster. Since only essential and required features are included in C, it saves processing power and improves speed.
- **Pointers.** C uses pointers, which improve performance by enabling direct interaction with the system memory.
- **Recursion.** C enables developers to backtrack by providing code reusability for every function.
- **Extensible.** A C program can be easily extended. If code is already written, new features and functionalities can be added to it with minor alterations.

C also comes with a few shortfalls, even though it's an ideal language for programming beginners due to its simple syntax, algorithms and modular structure. The following are a few disadvantages of using C:

- **OOP features.** C doesn't extend its support for object-oriented programming (OOP) features, which enables the creation of subclasses from parent classes.
  Unlike Java, Python or C++, multiple inheritances can't be created in C, which makes it difficult to reuse existing code.
- **Namespace feature.** C lacks namespace features, which means the same variable name can't be reused in one scope. Without namespaces, it's impossible to declare two variables with the same name.
- **Run-time checking.** C doesn't display code errors after each line of code; instead, all the errors are presented by the compiler after the program has been written. This can make code checking a challenge, especially for larger programs.
- **Exception handling.** C lacks exception handling, which is the ability to handle exceptions, such as bugs and anomalies that can happen during source code

- **Constructor and destructor.** Since C isn't object oriented, it doesn't offer constructor and destructor features. Constructing or destructing a variable in C must be done manually through a function or by other means.
- **Garbage collection.** C isn't equipped with garbage collection. This important feature automatically reclaims memory from objects that are no longer required by the library or an app.

Where is C used?

C has a wide range of real-world applications that aren't limited to the development of OSes and applications. C is also used in areas such as graphical user interface development and integrated development environments.

The following are some use cases for the C language:

- OSes, such as Unix and all Unix applications;
- databases, including Oracle Database, MySQL, Microsoft SQL Server and PostgreSQL, which are partially written in C;
- language compilers, including the C compiler;
- text editors;
- print spoolers;
- assemblers;
- network drivers;
- modern programs, such as Git and FreeBSD;
- language interpreters; and
- utilities, such as network drivers, mouse drivers and keyboard drivers.

Use of the Variables in C

Variables are the storage areas in a code that the program can easily manipulate. Every variable in C language has some specific type- that determines the layout and the size of the memory of the variable, the range of values that the memory can hold, and the set of operations that one can perform on that variable.

The name of a variable can be a composition of digits, letters, and also underscore characters. The name of the character must begin with either an underscore or a letter. In the case of C, the lowercase and uppercase letters are distinct. It is because C is case-sensitive in nature. Let us look at some more ways in which we name a variable.

Rules for Naming a Variable in C

We give a variable a meaningful name when we create it. Here are the rules that we must follow when naming it:

1. The name of the variable must not begin with a digit.

2. A variable name can consist of digits, alphabets, and even special symbols such as an underscore (_).

3. A variable name must not have any keywords, for instance, float, int, etc.

4. There must be no spaces or blanks in the variable name.

5. The C language treats lowercase and uppercase very differently, as it is case sensitive. Usually, we keep the name of the variable in the lower case.

Let us look at some of the examples,

int var1; // it is correct

int 1var; // it is incorrect – the name of the variable should not start using a number

int my_var1; // it is correct

int my$var // it is incorrect – no special characters should be in the name of the variable

char else; // there must be no keywords in the name of the variable

int my var; // it is incorrect – there must be no spaces in the name of the variable

int COUNT; // it is a new variable

int Count; // it is a new variable

int count; // it is a valid variable name

Data Type of the Variable

We must assign a data type to all the variables that are present in the C language. These define the type of data that we can store in any variable. If we do not provide the variable with a data type, the C compiler will ultimately generate a syntax error or a compile-time error.

The data Types present in the C language are float, int, double, char, long int, short int, etc., along with other modifiers.

Types of Primary/ Primitive Data Types in C Language

The variables can be of the following basic types, based on the name and the type of the variable:

Type of Variable	Name	Description	Uses
char	Character	It is a type of integer. It is typically one byte (single octet).	We use them in the form of single alphabets, such as X, r, B, f, k, etc., or for the ASCII character sets.
int	Integer	It is the most natural size of an integer used in the machine.	We use this for storing the whole numbers, such as 4, 300, 8000, etc.
float	Floating-Point	It is a floating-point value that is single precision.	We use these for indicating the real number values or decimal points, such as 20.8, 18.56, etc.
double	Double	It is a floating-point value that is double precision.	These are very large-sized numeric values that aren't really allowed in any data type that is a floating-point or an integer.
void	Void	It represents that there is an absence of type.	We use it to represent the absence of value. Thus, the use of this data type is to define various functions.

Let us look at a few examples,

```
// int type variable in C

int marks = 45;

// char type variable in C

char status = 'G';

// double type variable in C

double long = 28.338612;

// float type variable in C

float percentage = 82.5;
```

If we try to assign a variable with an incorrect value of datatype, then the compiler will (most probably) generate an error- the compile-time error. Or else, the compiler will convert this value automatically into the intended datatype of the available variable.

Let us look at an example,

```
#include <stdio.h>

int main() {

// assignment of the incorrect value to the variable

int a = 20.397;

printf(“Value is %d”, a);

return 0;

}
```

The output generated here will be:

20

As you can already look at this output- the compiler of C will remove the part that is present after the decimal. It is because the data types are capable of storing only the whole numbers.

We Cannot Change The Data Type

Once a user defines a given variable with any data type, then they will not be able to change the data type of that given variable in any case.

Let us look at an example,

```
// the int variable in C

int marks = 20;

float marks; // it generates an error
```

Variable Definition in C

The variable definition in C language tells the compiler about how much storage it should be creating for the given variable and where it should create the storage. Basically, the variable definition helps in specifying the data type. It contains a list of one variable or multiple ones as follows:

type variable_list;

In the given syntax, *type* must be a valid data type of C that includes w_char, char, float, int, bool, double, or any object that is user-defined. The *variable_list*, on the other hand, may contain one or more names of identifiers that are separated by commas. Here we have shown some of the valid declarations:

```
char c, ch;

int p, q, r;
```

double d;

float f, salary;

Here, the line int p, q, r; defines as well as declares the variables p, q, and r. It instructs the compiler to create three variables- named p, q, and r- of the type int.

We can initialize the variables in their declaration (assigned an initial value). The initializer of a variable may contain an equal sign- that gets followed by a constant expression. It goes like this:

type variable_name = value;

A few examples are −

extern int p = 3, q = 5; // for the declaration of p and q.

int p = 3, q = 5; // for the definition and initialization of p and q.

byte x = 22; // for the definition and initialization of x.

char a = 'a'; // the variable x contains the value 'a'.

In case of definition without the initializer: The variables with a static duration of storage are initialized implicitly with NULL (here, all bytes have a 0 value), while the initial values of all the other variables are not defined.

Declaration of Variable in C

Declaring a variable provides the compiler with an assurance that there is a variable that exists with that very given name. This way, the compiler will get a signal to proceed with the further compilation without needing the complete details regarding that variable.

The variable definition only has a meaning of its own during the time of compilation. The compiler would require an actual variable definition during the time of program linking.

The declaration of variables is useful when we use multiple numbers of files and we define the variables in one of the files that might be available during the time of program linking. We use the *extern* keyword for declaring a variable at any given place. Though we can declare one variable various times in a C program, we can only define it once in a function, a file, or any block of code.

Example

Let us look at the given example where we have declared the variable at the top and initialized and defined them inside the main function:

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

```
// Declaration of Variable

extern int p, q;

extern int c;

extern float f;

int main () {

/* variable definition: */

int p, q;

int r;

float i;

/* actual initialization */

p = 10;

q = 20;

r = p + q;

printf(“the value of r : %d \n”, r);

i = 70.0/3.0;

printf(“the value of i : %f \n”, i);

return 0;

}
```

The compilation and execution of the code mentioned above will produce the result as follows:

the value of r : 30

the value of i : 23.333334

Classification of Variables in C

The variables can be of the following basic types, based on the name and the type of the variable:

- **Global Variable**: A variable that gets declared outside a block or a function is known as a global variable. Any function in a program is capable of changing the value of a global variable. It means that the global variable will be available to all the functions in the code. Because the global variable in c is available to all the functions, we have to declare it at the beginning of a block. Explore, Global Variable in C to know more.
  Example,
  int value=30; // a global variable
  void function1(){
  int a=20; // a local variable
  }
- **Local Variable:** A local variable is a type of variable that we declare inside a block or a function, unlike the global variable. Thus, we also have to declare a local variable in c at the beginning of a given block.

Example,
void function1(){
int x=10; // a local variable
}
A user also has to initialize this local variable in a code before we use it in the program.

- **Automatic Variable:** Every variable that gets declared inside a block (in the C language) is by default automatic in nature. One can declare any given automatic variable explicitly using the keyword *auto*.
Example,
void main(){
int a=80; // a local variable (it is also automatic variable)
auto int b=50; // an automatic variable
}
- **Static Variable:** The static variable in c is a variable that a user declares using the *static* keyword. This variable retains the given value between various function calls.
Example, void function1(){

  int a=10; // A local variable

  static int b=10; // A static variable

  a=a+1;

  b=b+1;

  printf("%d,%d",a,b);

  }

  If we call this given function multiple times, then the local variable will print this very same value for every function call. For example, 11, 11, 11, and so on after this. The static variable, on the other hand, will print the value that is incremented in each and every function call. For example, 11, 12, 13, and so on.

- **External Variable:** A user will be capable of sharing a variable in multiple numbers of source files in C if they use an external variable. If we want to declare an external variable, then we need to use the keyword *extern*.
Syntax, extern int a=10;// external variable (also a global variable)

Rvalues and L values in C

There are two types of expressions in the C language:

rvalue − This term refers to the data value that gets stored at some type of memory address. The rvalue is basically an expression that has no value assigned to it. It means that an rvalue may appear on the RHS (right-hand side), but it cannot appear on the LHS (left-hand side) of any given assignment.

lvalue − lvalue expressions are the expressions that refer to any given memory location. The lvalue can appear as both- the RHS as well as the LHS of any assignment.

A variable is an lvalue. Thus, it may appear on the LHS of an assignment as well, along with the RHS. The numeric literals are rvalues. Thus, these might not be assigned. Thus, these can't appear on the LHS. Here are two statements- one valid and invalid:

Example,

int g = 10; // a valid statement

30 = 60; // an invalid statement; will generate a compile-time error

Practice Problems on Variables in C

1. Which of these is a primary data type in the C language?

1. char

2. float

3. void

4. enum

A. All of the above

B. 1, 3, 4

C. 1, 2, 4

D. 2, 3

Answer – A. All of the above

2. Which of these is the correct name for a variable?

1. int 1var;

2. int my$var

3. char else;

4. int my var;

5. int Count;

A. 3, 4

B. 5

C. 1, 2

D. All of the Above

E. None of the above

Answer – B. 5

3. The statement, “Every variable that gets declared inside a block (in the C language) is by default automatic in nature.” is:

A. True

B. False

C. Conditional

Answer – A. True

FAQs

Q1 Why do we declare a variable in a program?

Declaring a variable provides the compiler with an assurance that there is a variable that exists with that very given name. This way, the compiler will get a signal to proceed with the further compilation without needing the complete details regarding that variable.
The declaration of variables is useful when we use multiple numbers of files and we define the variables in one of the files that might be available during the time of program linking.

Q2 Why do we need to define any variable in a program?

The variable definition in C language tells the compiler about how much storage it should be creating for the given variable and where it should create the storage. Basically, the variable definition helps in specifying the data type. It contains a list of one variable or multiple ones as follows:
type variable_list;

Q3 **What would happen if we try to assign a variable with an incorrect value of datatype?**

If we try to assign a variable with an incorrect value of datatype, then the compiler will (most probably) generate an error- the compile-time error. Or else, the compiler will convert this value automatically into the intended datatype of the available variable.
Example,

```
#include
int main() {
// assignment of the incorrect value to the variable
int a = 20.397;
printf(“Value is %d”, a);
return 0;
}
```

The output generated here will be:
20
As you can already look at this output- the compiler of C will remove the part that is present after the decimal. It is because the data types are capable of storing only the whole numbers.

1.2 Character set data type and operator :

In the C programming language, the character set refers to a set of all the valid characters that we can use in the source program for forming words, expressions, and numbers.

The source character set contains all the characters that we want to use for the source program text. On the other hand, the execution character set consists of the set of those characters that we might use during the execution of any program. Thus, it is not a prerequisite that the execution character set and the source character set will be the same, or they will match altogether.

In this article, we will take a closer look at the Character Set in C according to the GATE Syllabus for CSE (Computer Science Engineering). Read ahead to know more.

Table of Contents

- Use Of Character Set In C
- Types Of Characters In C
  - Alphabets
  - Digits
  - Special Characters
  - White Spaces

Summary Of Special Characters In C

Purpose Of Character Set In C

  - Ascii Values
    - Control Characters
    - Printable Characters
    - Character Equivalence

Practice Problems On Character Set In C

FAQs

Use of Character Set in C

Just like we use a set of various words, numbers, statements, etc., in any language for communication, the C programming language also consists of a set of various different types of characters. These are known as the characters in C. They include digits, alphabets, special symbols, etc. The C language provides support for about 256 characters.

Every program that we draft for the C program consists of various statements. We use words for constructing these statements. Meanwhile, we use characters for constructing these statements. These characters must be from the C language character set. Let us look at the set of characters offered by the C language.

Types of Characters in C

The C programming language provides support for the following types of characters. In other words, these are the *valid* characters that we can use in the C language:

- Digits
- Alphabets
- Main Characters

All of these serve a different set of purposes, and we use them in different contexts in the C language.

Alphabets

The C programming language provides support for all the alphabets that we use in the English language. Thus, in simpler words, a C program would easily support a total of 52 different characters- 26 uppercase and 26 lowercase.

Type of Character	Description	Characters
Lowercase Alphabets	a to z	a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z
Uppercase Alphabets	A to Z	A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z

Digits

The C programming language provides the support for all the digits that help in constructing/ supporting the numeric values or expressions in a program. These range from 0 to 9, and also help in defining an identifier. Thus, the C language supports a total of 10 digits for constructing the numeric values or expressions in any program.

Type of Character	Description	Characters
Digits	0 to 9	0, 1, 2, 3, 4, 5, 6, 7, 8, 9

Special Characters

We use some special characters in the C language for some special purposes, such as logical operations, mathematical operations, checking of conditions, backspaces, white spaces, etc.

We can also use these characters for defining the identifiers in a much better way. For instance, we use underscores for constructing a longer name for a variable, etc.

The C programming language provides support for the following types of special characters:

Type of Character	Examples
Special Characters	` ~ @ ! $ # ^ * % & ( ) [ ] { } < > + = _ – | / \ ; : ‘ “ , . ?

White Spaces

The white spaces in the C programming language contain the following:

- Blank Spaces
- Carriage Return
- Tab
- New Line

Summary of Special Characters in C

Here is a table that represents all the types of character sets that we can use in the C language:

Type of Character	**Description**	**Characters**
Lowercase Alphabets	a to z	a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z
Uppercase Alphabets	A to Z	A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z
Digits	0 to 9	0, 1, 2, 3, 4, 5, 6, 7, 8, 9
Special Characters	–	` ~ @ ! $ # ^ * % & ( ) [ ] { } < > + = _ – | / \ ; : ‘ “ , . ?
White Spaces	–	Blank Spaces, Carriage Return, Tab, New Line

Purpose of Character Set in C

The character sets help in defining the valid characters that we can use in the source program or can interpret during the running of the program. For the source text, we have the source character set, while we have the execution character set that we use during the execution of any program.

But we have various types of character sets. For instance, one of the character sets follows the basis of the ASCII character definitions, while the other set consists of various kanji characters (Japanese).

The type of character set we use will have no impact on the compiler- but we must know that every character has different, unique values. The C language treats every character with different integer values. Let us know a bit more about the ASCII characters.

ASCII Values

All the character sets used in the C language have their equivalent ASCII value. The ASCII value stands for American Standard Code for Information Interchange value. It consists of less than 256 characters, and we can represent these in 8 bits or even less. But we use a special type for accommodating and representing the larger sets of characters. These are called the wide-character type or wchat_t.

However, a majority of the ANSI-compatible compilers in C accept these ASCII characters for both the character sets- the source and the execution. Every ASCII character will correspond to a specific numeric value.

Here is a list of all the ASCII characters, along with their assigned numeric values.

Control Characters

ASCII Value	Character	Meaning
0	NULL	Null
1	SOH	Start of Header
2	STX	Start of Text
3	ETX	End of Text
4	EOT	End of Transaction
5	ENQ	Enquiry
6	ACK	Acknowledgement
7	BEL	Bell
8	BS	Backspace

9	HT	Horizontal Tab
10	LF	Line Feed
11	VT	Vertical Tab
12	FF	Form Feed
13	CR	Carriage Return
14	SO	Shift Out
15	SI	Shift In
16	DLE	Data Link Escape
17	DC1	Device Control 1
18	DC2	Device Control 2
19	DC3	Device Control 3
20	DC4	Device Control 4
21	NAK	Negative Acknowledgement
22	SYN	Synchronous Idle
23	ETB	End of Trans Block
24	CAN	Cancel
25	EM	End of Medium

26	SUB	Substitute
27	ESC	Escape
28	FS	File Separator
29	GS	Group Separator
30	RS	Record Separator
31	US	Unit Separator

Printable Characters

ASCII Value	**Character**
32	Space
33	!
34	“
35	#
36	$
37	%
38	&
39	
40	(

41	)
42	≠
43	+
44	,
45	–
46	.
47	/
48	0
49	1
50	2
51	3
52	4
53	5
54	6
55	7
56	8
57	9

58	:
59	;
60	<
61	=
62	>
63	?
64	@
65	A
66	B
67	C
68	D
69	E
70	F
71	G
72	H
73	I
74	J

75	K
76	L
77	M
78	N
79	O
80	P
81	Q
82	R
83	S
84	T
85	U
86	V
87	W
88	X
89	Y
90	Z
91	[

92	|
93	]
94	^
95	_
96	`
97	a
98	b
99	c
100	d
101	e
102	f
103	g
104	h
105	i
106	j
107	k
108	l

109	m
110	n
111	o
112	p
113	q
114	r
115	s
116	t
117	u
118	v
119	w
120	x
121	y
122	z
123	{
124	|
125	}

126	〃
127	DEL

(DEL is also a control character.)

Character Equivalence

Here are all the character sets in ASCII. The table below displays all the character's hexadecimal, decimal, and octal values:

Character	Oct	Dec	Hex
\0	00	0	0x0
\001	01	1	0x1
\002	02	2	0x2
\003	03	3	0x3
\004	04	4	0x4
\005	05	5	0x5
\006	06	6	0x6
\007	07	7	0x7
\b	010	8	0x8
\t	011	9	0x9
\n	012	10	0xA

\v	013	11	0xB
\f	014	12	0xC
\r	015	13	0xD
\016	016	14	0xE
\017	017	15	0xF
\020	020	16	0x10
\021	021	17	0x11
\022	022	18	0x12
\023	023	19	0x13
\024	024	20	0x14
\025	025	21	0x15
\026	026	22	0x16
\027	027	23	0x17
\030	030	24	0x18
\031	031	25	0x19
\032	032	26	0x1A
\033	033	27	0x1B

\034	034	28	0x1C
\035	035	29	0x1D
\036	036	30	0x1E
\037	037	31	0x1F
(space)	040	32	0x20
!	041	33	0x21
“	042	34	0x22
#	043	35	0x23
$	044	36	0x24
%	045	37	0x25
&	046	38	0x26
\	047	39	0x27
(	050	40	0x28
)	051	41	0x29
	052	42	0x2A
+	053	43	0x2B
,	054	44	0x2C

_	055	45	0x2D
.	056	46	0x2E
/	057	47	0x2F
0	060	48	0x30
1	061	49	0x31
2	062	50	0x32
3	063	51	0x33
4	064	52	0x34
5	065	53	0x35
6	066	54	0x36
7	067	55	0x37
8	070	56	0x38
9	071	57	0x39
|	072	58	0x3A
;	073	59	0x3B
<	074	60	0x3C
=	075	61	0x3D

>	076	62	0x3E
?	077	63	0x3F

Character	Oct	Dec	Hex
__	0100	64	0x40
A	0101	65	0x41
B	0102	66	0x42
C	0103	67	0x43
D	0104	68	0x44
E	0105	69	0x45
F	0106	70	0x46
G	0107	71	0x47
H	0110	72	0x48
I	0111	73	0x49
J	0112	74	0x4A
K	0113	75	0x4B
L	0114	76	0x4C
M	0115	77	0x4D

N	0116	78	0x4E
O	0117	79	0x4F
P	0120	80	0x50
Q	0121	81	0x51
R	0122	82	0x52
S	0123	83	0x53
T	0124	84	0x54
U	0125	85	0x55
V	0126	86	0x56
W	0127	87	0x57
X	0130	8	0x58
Y	0131	89	0x59
Z	0132	90	0x5A
[	0133	91	0x5B
\	0134	92	0x5C
]	0135	93	0x5D
^	0136	94	0x5E

_	0137	95	0x5F
‘	0140	96	0x60
a	0141	97	0x61
b	0142	98	0x62
c	0143	99	0x63
d	0144	100	0x64
e	0145	101	0x65
f	0146	102	0x66
g	0147	103	0x67
h	0150	104	0x68
i	0151	105	0x69
j	0152	106	0x6A
k	0153	107	0x6B
l	0154	108	0x6C
m	0155	109	0x6D
n	0156	110	0x6E
o	0157	111	0x6F

p	0160	112	0x70
q	0161	113	0x71
r	0162	114	0x72
s	0163	115	0x73
t	0164	116	0x74
u	0165	117	0x75
v	0166	118	0x76
w	0167	119	0x77
x	0170	120	0x78
y	0171	121	0x79
z	0172	122	0x7A
{	0173	123	0x7B
|	0174	124	0x7C
}	0175	125	0x7D
~	0176	126	0x7E
\177	0177	127	0x7F

Practice Problems on Character Set in C

1. Which of these is a type of character set used in the C language?

A. Digits

B. Alphabets Characters

D. All of the above

Answer – D. All of the above

2. What types of alphabets does the language support?

A. Lowercase Alphabets and Characters

B. Uppercase Alphabets and Characters

C. All of the above

Answer – C. All of the above

3. How many characters does the C programming language support in total?

A. 52

B. 26

C. 256

D. 86

Answer – C. 256

4. How many digits do the C programming language support as character sets?

A. Nine

B. Eight

C. Five

D. Ten

Answer – D. Ten

FAQs

Q1 What constitutes the white spaces in the C language?

In the C programming language, the white spaces contain the following:

- Blank Spaces

- Carriage Return
- Tab
- New Line

Q2 What are ASCII values in C?

All the character sets used in the C language have their equivalent ASCII value. The ASCII value stands for American Standard Code for Information Interchange value. It consists of less than 256 characters, and we can represent these in 8 bits or even less.

However, a majority of the ANSI-compatible compilers in C accept these ASCII characters for both the character sets- the source and the execution. Every ASCII character will correspond to a specific numeric value.

Q3 What is wchat_t?

The ASCII consists of less than 256 characters, and we can represent these in 8 bits or even less. But we use a special type for accommodating and representing the larger sets of characters. These are called the wide-character type or wchat_t.

Q4 **What is the use of special characters if we have digits in the C language?**

We use some special characters in the C language for some special purposes, such as logical operations, mathematical operations, checking of conditions, backspaces, white spaces, etc.

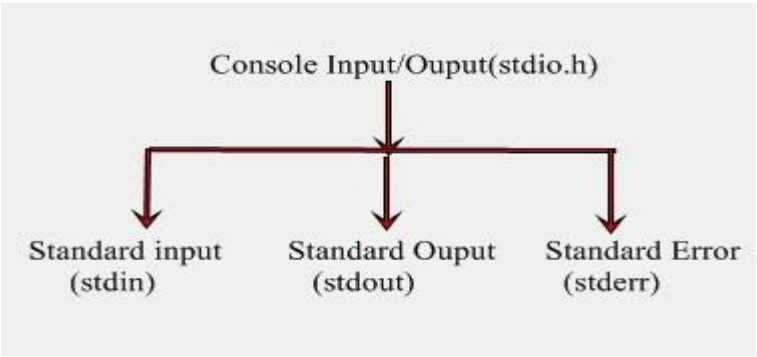
We can also use these characters for defining the identifiers in a much better way. For instance, we use underscores for constructing a longer name for a variable, etc.

1.3 Input and Output Statement:

C Input Output Statements

by admin

In C Language input and output function are available as C compiler functions or C libraries provided with each C compiler implementation. These all functions are collectively known as **Standard I/O Library function**. Here I/O stands for Input and Output used for different inputting and outputting statements. These I/O functions are categorized into three processing functions. Console input/output function (deals with keyboard and monitor), disk input/output function (deals with floppy or hard disk), and port input/output function (deals with a serial or parallel port). As all the input/output statements deals with the console, so these are also **Console Input/Output functions**. Console Input/Output function access the three major files before the execution of a C Program. These are as follows:



- **stdin**: This file is used to receive the input (usually is keyborad file, but can also take input from the disk file).
- **stdout**: This file is used to send or direct the output (usually is a monitor file, but can also send the output to a disk file or any other device).
- **stderr**: This file is used to display or store error messages.

Input Ouput Statement

Input and Output statement are used to read and write the data in C programming. These are embedded in stdio.h (standard Input/Output header file).

Input means to provide the program with some data to be used in the program and **Output** means to display data on screen or write the data to a printer or a file.C programming language provides many built-in functions to read any given input and to display data on screen when there is a need to output the result.

There are mainly two of Input/Output functions are used for this purpose. These are discussed as:

- Unformatted I/O functions
- Formatted I/O functions

Unformatted I/O functions

There are mainly six unformatted I/O functions discussed as follows:

- getchar()
- putchar()
- gets()
- puts()
- getch()
- getche()

getchar()

This function is an Input function. It is used for reading a single character from the keyboard. It is a buffered function. Buffered functions get the input from the keyboard and store it in the memory buffer temporally until you press the Enter key.

The general syntax is as:

```
v = getchar();
```
where v is the variable of character type. For example:

```
char n;
n = getchar();
```
A simple C-program to read a single character from the keyboard is as:

```
/*To read a single character from the keyboard using the getchar() function*/
#include <stdio.h>
main()
{
char n;
n = getchar();
}
```
putchar()

This function is an output function. It is used to display a single character on the screen. The general syntax is as:

```
putchar(v);
```
where v is the variable of character type. For example:

```
char n;
putchar(n);
```
A simple program is written as below, which will read a single character using getchar() function and display inputted data using putchar() function:

```
/*Program illustrate the use of getchar() and putchar() functions*/
#include <stdio.h>
main()
{
char n;
n = getchar();
putchar(n);
}
```
gets()

This function is an input function. It is used to read a string from the keyboard. It is also a buffered function. It will read a string when you type the string from the keyboard and press the Enter key from the keyboard. It will mark null character (‘\0’) in the memory at the end of the string when you press the enter key. The general syntax is as:

```
gets(v);
```
where v is the variable of character type. For example:

```
char n[20];
gets(n);
```
A simple C program to illustrate the use of gets() function:

```
/*Program to explain the use of gets() function*/
#include <stdio.h>
main()
```

```
{
char n[20];
gets(n);
}
```
puts()

This is an output function. It is used to display a string inputted by gets() function. It is also used to display a text (message) on the screen for program simplicity. This function appends a newline (“\n”) character to the output.

The general syntax is as:

```
puts(v);
```
or

```
puts("text line");
```
where v is the variable of character type.

A simple C program to illustrate the use of puts() function:

```
/*Program to illustrate the concept of puts() with gets() functions*/
#include <stdio.h>
main()
{
char name[20];
puts("Enter the Name");
gets(name);
puts("Name is :");
puts(name);
}
```
The Output is as follows:

```
Enter the Name
Geek
Name is:
Geek
```
getch()

This is also an input function. This is used to read a single character from the keyboard like getchar() function. But getchar() function is a buffered is function, getchar() function is a non-buffered function. The character data read by this function is directly assigned to a variable rather it goes to the memory buffer, the character data is directly assigned to a variable without the need to press the Enter key.

Another use of this function is to maintain the output on the screen till you have not press the Enter Key. The general syntax is as:

```
v = getch();
```
where v is the variable of character type.

A simple C program to illustrate the use of getch() function:

```
/*Program to explain the use of getch() function*/
#include <stdio.h>
main()
{
char n;
puts("Enter the Char");
n = getch();
puts("Char is :");
putchar(n);
getch();
}
```

The output is as follows:

```
Enter the Char
Char is L
```

getche()

All are same as getch(0 function execpt it is an echoed function. It means when you type the character data from the keyboard it will visible on the screen. The general syntax is as:

```
v = getche();
```

where v is the variable of character type.

A simple C program to illustrate the use of getch() function:

```
/*Program to explain the use of getch() function*/
#include <stdio.h>
main()
{
char n;
puts("Enter the Char");
n = getche();
puts("Char is :");
putchar(n);
getche();
}
```

The output is as follows:

```
Enter the Char L
Char is L
```

Formatted I/O functions

Formatted I/O functions which refers to an Input or Ouput data that has been arranged in a particular format. There are mainly two formatted I/O functions discussed as follows:

- scanf()
- printf()

scanf()

The scanf() function is an input function. It used to read the mixed type of data from keyboard. You can read integer, float and character data by using its control codes or format codes. The general syntax is as:

```
scanf("control strings",arg1,arg2,.............argn);
```

or

```
scanf("control strings",&v1,&v2,&v3,.................&vn);
```

Where arg1,arg2,…….argn are the arguments for reading and v1,v2,v3,……..vn all are the variables.

The scanf() format code (spedifier) is as shown in the below table:

Format Code	**Meaning**
%c	To read a single character
%d	To read a signed decimal integer (short)
%ld	To read a signed long decimal integer
%e	To read a float value exponential
%f	To read a float (short0 or a single precision value
%lf	To read a double precision float value
%g	To read double float value

%h	To read short integer
%i	To read an integer (decimal, octal, hexadecimal)
%o	To read an octal integer only
%x	To read a hexadecimal integer only
%u	To read unsigned decimal integer (used in pointer)
%s	To read a string
%[..]	To read a string of words from the defined range
%[^]	To read string of words which are not from the defined range

Example Program:

```
/*Program to illustrate the use of formatted code by using the formatted scanf()
function */
#include <stdio.h>
main()
{
char n,name[20];
int abc;
float xyz;
printf("Enter the single character, name, integer data and real value");
scanf("\n%c%s%d%f", &n,name,&abc,&xyz);
getch();
}
```

printf()

This ia an output function. It is used to display a text message and to display the mixed type (int, float, char) of data on screen. The general syntax is as:

```
printf("control strings",&v1,&v2,&v3,.....&vn);
```

or

```
printf("Message line or text line");
```
Where v1,v2,v3,……..vn all are the variables.

The control strings use some printf() format codes or format specifiers or conversion characters. These all are discussed in the below table as:

Format Code	**Meaning**
%c	To read a single character
%s	To read a string
%d	To read a signed decimal integer (short)
%ld	To read a signed long decimal integer
%f	To read a float (short0 or a single precision value
%lf	To read a double precision float value
%e	To read a float value exponential
%g	To read double float value
%o	To read an octal integer only
%x	To read a hexadecimal integer only

%u	To read unsigned decimal integer (used in pointer)

Example Program:

```
/*Below the program which show the use of printf() function*/
#include <stdio.h>
main()
{
int a;
float b;
char c;
printf("Enter the mixed type of data");
scanf("%d",%f,%c",&a,&b,&c);
getch();
}
```

1.4 Jumping and Branching Statement is X statemaent Looping Statement:

How to Use Jump Statement in C? & Its Types | DataTrained

By Chandrakishor Gupta

April 20, 2023

Introduction

Jump Statement in C is used in C programming to transfer the program control from one part of the code to another. C provides three types of Jump Statements in C, namely, break, continue, and goto. The break statement is used to terminate the execution of a loop or switch statement. When the break statement is encountered, the program control jumps to the statement following the loop or switch.

The continue statement is used to skip the remaining code in the current iteration of a loop and move to the next iteration. When the continue statement is encountered, the program control jumps to the beginning of the loop.

The goto statement is used to transfer the program control to a labeled statement in the same function. This statement is often criticized for its potential to create unstructured and hard-to-***maintain code***.

Jump Statement in C provides programmers with a way to alter the normal flow of execution in a program. However, it is important to use them judiciously as they can make code difficult to read and debug. In most cases, it is recommended to use structured programming constructs such as loops and conditional statements to control the flow of execution.

Types of Jump Statements in C

In C programming language, there are three types of Jump Statement in C that allow the program to jump to a different part of the code. These Jump Statement in C are:

break statement: The break statement is used to terminate a loop or switch statement. When the break statement is encountered, the control is transferred to the statement immediately following the loop or switch.

continue statement: The continue statement is used to skip the current iteration of a loop and continue with the next iteration. When the continue statement is encountered, the control is transferred to the beginning of the loop.

goto statement: The goto statement is used to transfer the control to a labeled statement in the code. The labeled statement is identified by a label followed by a colon. The use of the goto statement is discouraged as it can make the code difficult to read and maintain.

These Jump Statements in C are useful in controlling the flow of the program and can be used to optimize the code. However, their use should be limited to situations where it is necessary, as excessive use of Jump Statements in C can make the code difficult to understand and maintain. It is important to use these statements judiciously and only when they are absolutely necessary.

How to Use the ‘Break’ Statement in C?

In C programming, the ‘break’ statement is a control flow statement that is used to terminate a loop or a switch statement. The break statement is useful when you want to exit from a loop or switch statement before it has finished executing all iterations.

The syntax for using the ‘break’ statement in C is simple. **Jump statements in C++,** The ‘break’ keyword is used inside a loop or switch statement. When the ‘break’ statement is encountered, the control flow jumps to the end of the loop or switch statement, and execution continues with the next statement after the loop or switch.

For example, suppose you have a while loop that iterates over a list of numbers and you want to exit the loop when you find a number that is greater than 10. You can use the ‘break’ statement as follows:

```
css

while (num_list[i] < 10) {

  if (num_list[i] > 10) {

    break;

  }

  i++;

}
```

In this example, the loop will continue iterating as long as the value of ‘num_list[i]’ is less than 10. When the ‘if’ condition is true, the ‘break’ statement is executed, and the control flow jumps to the end of the loop, bypassing any remaining iterations.

Similarly, the ‘break’ statement can be used in a switch statement to exit the switch block prematurely. When the ‘break’ statement is encountered in a switch block, the control flow jumps to the end of the block, bypassing any remaining cases.

Overall, the ‘break’ statement is a powerful tool in C programming, allowing you to control the flow of execution in loops and switch statements. It can help you write more efficient and concise code by allowing you to exit loops or switch blocks as soon as a particular condition is met.

The ‘continue’ Statement and Its Usage in C

In C programming language, the ‘continue’ statement is a flow control statement that is used within loops to skip the remaining statements in the current iteration of the loop and move on to the next iteration of the loop.

The ‘continue’ statement is typically used within loops such as ‘for’ and ‘while’ loops to skip certain iterations based on certain conditions. When the ‘continue’ statement is executed within a loop, the control flow of the program jumps back to the beginning of the loop, and the loop’s condition is checked again to determine if another iteration should be performed.

For example, consider the following ‘for’ loop:

c

Copy code

```
for(int i=0; i<10; i++){
    if(i == 5){
        continue;
    }
    printf(“%dn”, i);
}
```

In this code snippet, the loop will iterate from i=0 to i=9, but when i=5, the ‘continue’ statement will skip the remaining statements within the loop and jump to the next iteration. As a result, the value 5 will not be printed, and the output of the loop will be:

Code

```
0
```

1

2

3

4

6

7

8

9

In summary, the ‘continue’ statement in C allows you to skip the remaining statements within a loop and move on to the next iteration based on certain conditions. This can be useful in situations where you want to skip certain iterations of a loop without exiting the loop entirely.

Understanding the ‘goto’ Statement in C

In C programming, the ‘goto’ statement is used to transfer control to a specified label within the same function. It allows you to jump to a particular section of code, bypassing any statements that appear in between.

The syntax of the ‘goto’ statement is as follows:

Arduino

Copy code

goto label;

Here, ‘label’ is a user-defined identifier followed by a colon (:), and it represents the target location to which control is transferred.

For example, consider the following code:

Perl

int x = 0;

start:

printf(“The value of x is %dn”, x);

```
    x++;

    if (x < 10)

        goto start;
```

Here, the label ‘start:’ is defined at the beginning of the code block. The code inside the block is executed, and then the ‘goto start’ statement transfers control back to the ‘start’ label. This process continues until the value of ‘x’ becomes greater than or equal to 10.

While the ‘goto’ statement can be a useful tool in some cases, it should be used sparingly and with caution. It can make code more difficult to read and understand, and it can also lead to hard-to-find bugs if used improperly.

In summary, the ‘goto’ statement is a C language construct that allows you to transfer control to a labeled section of code within the same function. It can be a powerful tool in some cases, but it should be used judiciously and with caution.

When to Use the ‘return’ Statement in C

In C, the return statement is used to exit a function and return a value to the calling function. It is a powerful statement that allows you to control the flow of a program and make it more efficient.

The return statement is often used to pass the result of a function back to the calling function. When a function is called, it performs its task and then returns a value. This value can then be used by the calling function to perform further computations or to make decisions based on the result.

In addition to returning a value, the return statement can also be used to exit a function prematurely. For example, if an error condition is encountered in a function, the function can be terminated immediately using a return statement. This helps to prevent further damage to the program or data.

It is important to note that the return statement can only be used within a function. It cannot be used in the main program. Additionally, if a function returns a value, it must be specified in the function header.

In summary, a return statement is a powerful tool in C programming that allows you to control the flow of your program and pass results between functions. It is essential for writing efficient, error-free code.

Examples of Jump Statement in C

In ***C programming language***, Jump Statements in C is used to transfer control from one part of the code to another part. There are three types of Jump Statement in C: break, continue, and goto.

Break: The break statement is used to exit from a loop or a switch statement. When the break statement is encountered in a loop, the loop terminates and control is transferred to the next statement after the loop. In a switch statement, the break statement is used to terminate the switch and transfer control to the next statement after the switch.

Continue: The continue statement is used to skip the current iteration of a loop and move to the next iteration. When the continue statement is encountered in a loop, the remaining statements in the loop for that iteration are skipped, and control is transferred to the next iteration of the loop.

Goto: The goto statement is used to transfer control to a labeled statement within the same function. It allows the programmer to jump to a specific part of the code regardless of the normal control flow. The use of the goto statement is generally discouraged as it can make code difficult to read and maintain.

In conclusion, Jump Statement

in C are a useful tool for controlling the flow of a program and can be used to optimize code and make it more efficient. However, it is important to use them judiciously and with caution to ensure that the code remains readable and maintainable.

Common Mistakes to Avoid When Using Jump Statements in C

Jump Statement in C is used in C programming language to transfer control from one part of the program to another. There are three types of Jump Statements in C: break, continue, and goto. While these statements can be useful in certain situations, they can also lead to errors if not used properly. Here are some common mistakes to avoid when using Jump Statement in C:

Using break or continue in nested loops: When using break or continue in nested loops, it can be easy to accidentally break out of the wrong loop or skip over important code. To avoid this, be sure to label your loops and specify which loop you want to break out of or continue.

Using goto: While goto can be useful in certain situations, it is generally considered bad practice to use it. Goto statements can make code harder to read and debug and can lead to spaghetti code.

Using Jump Statement in C inside switch statements: Using break or continue inside a switch statement can also lead to errors, as it can cause the switch statement to terminate prematurely or skip over important code. Instead, use a flag variable to control the flow of the program.

Using Jump Statement in C to exit functions: Using break or continue to exit a function can also cause errors, as it can leave the function in an undefined state. Instead, use the return to exit the function.

In summary, it is important to use Jump Statement in C judiciously in C programming, and to avoid common mistakes such as using them in nested loops, using goto, using them inside switch statements, and using them to exit functions. By following these guidelines, you can write clean, error-free code that is easy to read and maintain.

Click here to about more about: data science in India

Best Practices for Using Jump Statements in C

Jump stat in C are used to control the flow of execution of a program. They allow the program to skip over certain sections of code, or to repeat sections of code, depending on certain conditions. However, Jump Statement in C should be used with care, as they can make the code harder to read and understand, and can also introduce subtle bugs if used improperly.

Here are some best practices for using Jump Statement in C:

Use Jump Statements in C sparingly. In general, it's better to use conditional statements (like if/else statements) or loops (like for/while loops) to control the flow of execution, rather than relying on Jump Statement in C. Jump Statement in C should only be used when it's absolutely necessary to skip over code or repeat code.

Always label your loops and switch statements. When using Jump Statement in C like break and continue inside loops and switch statements, it's important to label them so that you can refer to them later in your code. This makes the code more readable and easier to understand.

Avoid using goto statements. The goto statement is a powerful jump statement that can be used to transfer control to any point in the code, but it's also very easy to abuse. In general, it's better to use other flow control statements, like loops and conditional statements, rather than goto.

Don't mix Jump Statement in C with complex expressions. Jump Statement in C like to break and continue should be used with simple expressions that are easy to understand. If you start mixing them with complex expressions or nested loops, the code can quickly become hard to read and understand.

Use comments to explain the purpose of the Jump Statement in C. If you do use Jump Statement in C in your code, be sure to include comments that explain why they are necessary and what they are doing. This will make the code easier to understand for anyone who needs to read or modify it in the future.

Read about: ***data analyst course in pune***

Alternatives to Jump Statement in C

In C programming, Jump Statement in C like "goto", "break" and "continue" can help to control the flow of code execution. However, using these statements can make the code more difficult to understand and maintain. Fortunately, there are several alternatives to Jump Statement in C that can be used to achieve the same effect in a more readable and organized way.

One alternative is to use Boolean expressions or flags to control loops or conditional statements. This can make the code more intuitive and easier to follow. For example, instead of using a "goto" statement to jump out of a loop, a Boolean flag can be set to exit the loop in a controlled manner.

Another alternative is to use functions to encapsulate complex behavior. By breaking up code into smaller, more manageable functions, it is easier to control the flow of execution without the need for Jump Statement in C.

In addition, structured programming techniques like "structured if" statements and "structured loops" can help to reduce the need for Jump Statements in C. These techniques involve using nested conditional statements and loops to create a more organized and structured approach to programming.

Overall, while Jump Statement in C can be useful in certain situations, it is generally better to avoid them in favor of more structured and readable code. By using Boolean expressions, functions, and structured programming techniques, it is possible to achieve the same control flow without sacrificing code quality.

Conclusion

In conclusion, Jump Statement in C is a powerful tool in the C programming language that allows programmers to control the flow of execution of a program. The three Jump Statement in C available in C are the break, continue, and goto statements.

The break statement is used to immediately exit a loop or switch statement, while the continue statement is used to skip the current iteration of a loop and move on to the next one. The goto statement allows a programmer to transfer control to any point in the program.

While these statements can be useful in certain situations, they can also make code difficult to read and understand, and their use should be limited. It is important to use them judiciously and to consider alternative approaches when possible.

Overall, understanding how to use Jump Statement in C effectively can help programmers write more efficient and readable code in C.

Frequently Asked Questions

1. What is a jump statement in C?

A jump statement is a keyword in the C programming language that is used to transfer control from one part of the program to another.

2. What are the types of Jump Statement in C?

There are three types of Jump Statement in C – break, continue, and goto.

3. What is the purpose of the break statement in C?

The break statement in C is used to exit a loop or switch statement before its normal termination.

4. What is the purpose of the continue statement in C?

The continue statement in C is used to skip over the current iteration of a loop and proceed with the next iteration.

5. What is the purpose of the goto statement in C?

The goto statement in C is used to transfer control to a labeled statement within the same function. However, its use is generally discouraged as it can lead to spaghetti code and make the program difficult to understand and debug.

Branching Statements

Overview

A branch is an instruction in a computer program that can cause a computer to begin executing a different instruction sequence and thus deviate from its default behavior of executing instructions in order.[1] Common branching statements include `break`, `continue`, `return`, and `goto`.

Discussion

Branching statements allow the flow of execution to jump to a different part of the program. The common branching statements used within other control structures include: `break`, `continue`, `return`, and `goto`. The goto is rarely used in modular structured programming. Additionally, we will add to our list of branching items a pre-defined function commonly used in programming languages of: `exit`.

Examples

break

The break is used in one of two ways; with a switch to make it act like a case structure or as part of a looping process to break out of the loop. The following gives the appearance that the loop will execute 8 times, but the break statement causes it to stop during the fifth iteration.

```
counter = 0;
While counter < 8
    Output counter
    If counter == 4
        break
    counter += 1
```

continue

The following gives the appearance that the loop will print to the monitor 8 times, but the continue statement causes it not to print number 4.

```
For counter = 0, counter < 8, counter += 1
    If counter == 4
        continue
    Output counter
```

return

The return statement exits a function and returns to the statement where the function was called.

```
Function DoSometing

    statements

Return <optional return value>
```

goto

The goto structure is typically not accepted in good structured programming. However, some programming languages allow you to create a label with an identifier name followed by a colon. You use the command word `goto` followed by the label.

```
some lines of code;

goto label;                 // jumps to the label

some lines of code;

some lines of code;

some lines of code;

label: some statement;      // Declared label

some lines of code;
```

exit

Although exit is technically a pre-defined function, it is covered here because of its common usage in programming. A good example is the opening a file and then testing to see if the file was actually opened. If not, we have an error that usually indicates that we want to prematurely stop the execution of the program. The exit function terminates the running of the program and in the process returns an integer value back to the operating system. It fits the definition of branching which is to jump to some other place in the program.

Key Terms

branching statements

Allow the flow of execution to jump to a different part of the program.

break

A branching statement that terminates the existing structure.

continue

A branching statement that causes a loop to stop its current iteration and begin the next one.

exit

A predefined function used to prematurely stop a program and return to the operating system.

goto

An unstructured branching statement that causes the logic to jump to a different place in the program.

return

A branching statement that causes a function to jump back to the function that called it.

Branching Statements in C

The C programming language is a procedural programming language and executes the program in a sequential manner from top to down. But sometimes we need to skip some lines of code according to some of our requirements. For this, we have Branching Statements in C. Here we are going to discuss different types of branching statements in C along with examples of each statement.

Branching Statements in C

Branching Statements in C are those statements in C language that helps a programmer to control the flow of execution of the program according to the requirements. These Branching Statements are considered an essential aspect of the Programming Languages and are essentially used for executing the specific segments of code on basis of some conditions.

Types of Branching Statements in C

The Branching Statements in C are categorized as follows.

- Conditional Branching Statements in C
- if Statement
- else Statement
- else-if Statement
- switch Statement

- Unconditional Branching Statements in C
- goto Statement
- break Statement
- continue Statement

All of these statements are explained below in detail. So, let's start.

Conditional Branching Statements in C

Conditional Branching Statements in C are used to execute the specific blocks of code on the basis of some condition (as per requirements). These type of Branching statements in C enables the programmers to execute the code only and only certain statements are met. The following are different types of conditional branching statements in C.

- if statement
- else statement
- else-if statement
- switch statement

if Statement

This statement is used to execute the specific block of code if a certain condition is evaluated to be true.

Syntax of if Statement in C

Here is the syntax of the if statement in C.

```
if (condition) {

    //code to be executed if condition specified evaluates is true

}
```

Here in the above syntax "condition" is a logical expression that is evaluated for being true or false. If the condition is evaluated to be true, the code within curly braces will be executed but if the condition is false, the code in braces will be skipped.

Example of if Statement in C

Here is an example demonstrating the if statement in action.

- C

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

```
int main() {

int x = 10;

if (x > 5) {

printf("x is greater than 5");

}

return 0;

}
```

Output

```
x is greater than 5
```

Explanation:

In the above example, the "if" statement checks whether the value of "x" is greater than 5. Since the condition is true (x is indeed greater than 5), the code within the curly braces is executed, and the message "x is greater than 5" is printed to the console.

else Statement

The else Statement in C is considered just the opposite of the if statement. This statement is used to execute the code if the condition specified in the if statement evaluates to false.

Syntax of else Statement in C

Here is the syntax of the else Statement.

```
if (condition) {

    // statemnets

} else{

    // code executed if condition is false

}
```

Example of else Statement in C

Here is an example of how the else statement works in C.

- C

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

int x = 10;
```

```
if (x < 5) {

printf("x is less than 5\n");

} else {

printf("x is greater than or equal to 5\n");

}

return 0;

}
```

Output

```
x is greater than or equal to 5
```

Explanation:

In this example, the if statement in C checks whether the value of the variable x is less than 5 or not. Since the value of x (which is 10), this condition is false, and the code within the "if" block is not executed. Instead, the code within the "else" block is executed, which prints the message "x is greater than or equal to 5" to the console.

else if Statement

The else if statement in Cis used when we want to check multiple conditions. It follows an "if" statement and is executed if the previous "if" statement's condition is false.

Syntax of else-if Statement in C

Here is the syntax for the "else if" statement:

```
if (condition1) {

    // code to be executed if condition1 is true

}

else if (condition2) {

    // code to be executed if condition2 is true and condition1 is false

}
```

The "else if" statement can be repeated multiple times to check for multiple conditions. If all conditions are false, the code within the final "else" block will be executed.

Example of else-if Statement in C

Here is an example of the "else if" statement in action.

- C

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

int x = 10;

if (x > 15) {

printf("x is greater than 15");

}

else if (x > 5) {

printf("x is greater than 5 but less than or equal to 15");

}

else {

printf("x is less than or equal to 5");

}

return 0;

}
```

Output

```
x is greater than 5 but less than or equal to 15
```

Explanation:

In the above code, we have used a complete block of if, else if and else statements. The if statement first checks whether the value of x is greater than 15 or not. Since x(which is 10) is less than 15, so the control moves to the else if block. Now the program checks whether x is greater than 5 or not. This time the condition is true, so the code block inside this else-if block will be executed and we get the required output on the screen i.e., "x is greater than 5 but less than or equal to 15".

switch Statement

The switch statement in C is used when we have multiple conditions to check. It is often used as an alternative to multiple "if" and "else if" statements.

Syntax of switch Statement in C

The syntax for the switch statement is as follows:

```
switch (expression) {

    case value1:
```

```
        // code to be executed if expression equals value1

        break;

    case value2:

        // code to be executed if expression equals value2

        break;

    ...

    default:

        // code to be executed if none of the cases match

        break;

}
```

The "expression" in the above syntax is the value being evaluated. Each "case" statement represents a possible value of the expression. If the expression matches one of the "case" statements, the code block within the matching "case" statement is executed. If none of the "case" statements match, the code within the "default" block is executed.

Example of switch Statement in C

Here is an example how the switch statement in C works.

- C

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

int x = 2;

switch (x) {

case 1:

printf("x is 1");

break;

case 2:

printf("x is 2");

break;

case 3:
```

```
printf("x is 3");

break;

default:

printf("x is not 1, 2, or 3");

break;

}

return 0;

}
```

Output

```
x is 2
```

Explanation:
In the above example, the "switch" statement checks the value of "x". Since "x" is equal to 2, the code within the second "case" statement is executed, and the message "x is 2" is printed to the console.

Unconditional Branching Statements in C

Unconditional branching statements are used in C to change the normal flow of program execution. These statements allow programmers to jump to a specific point in their code regardless of any condition.

There are the following types of unconditional branching statements in C.

- goto Statement
- break Statement
- continue Statement

Let us discuss these in detail one by one.

goto Statement

The "goto" statement is an unconditional branching statement that allows programmers to jump to a specific labeled statement within their code.

Syntax of goto Statement in C

Here is the syntax for the "goto" statement:

```
goto label;
```

```
...



label:

    // code to be executed
```

The "goto" statement jumps to the statement labeled "label" and executes the code within that block. The "label" can be any valid identifier followed by a colon (:).

Example of goto Statement in C

Here is an example of the goto statement.

- C

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

int x = 0;

start:

x++;

if (x < 10) {

goto start;

}

printf("x is %d", x);

return 0;

}
```

Output

```
x is 10
```

Explanation:

In the above example, the program uses a goto statement to jump back to the "start" label after incrementing "x". The if statement checks whether "x" is less than 10, and if it is, the program jumps back to the "start" label. This process continues until "x" is equal to 10, at which point the program exits the loop and prints "x is 10" to the console output.

break Statement

The break Statement is one of the unconditional Branching Statements in C which is generally used in the loops for exiting the loop before the completion of the loop.

Syntax of break Statement in C

Here is the syntax for the "break" statement in C.

```
break;
```

When we use the above statement within any loop, it simply exits the loop regardless of the completion of loop.

Example of break Statement in C

Here is an example of the break statement in C within a while loop:

- C

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

int x = 0;

while (x < 10) {

x++;

if (x == 5) {

break;

}

}

printf("x is %d", x);

return 0;

}
```

Output

```
x is 5
```

Explanation:

In the above example, the program uses a "while" loop to increment "x" until it is equal to 5. When "x" is equal to 5, the "break" statement is executed, causing the program to exit the loop prematurely. The final output is "x is 5".

continue Statement

The continue statement is used in C programming language to skip the current iteration of a loop and move to the next iteration. This statement is typically used in loops like for or while.

Syntax of continue Statement in C

continue Statement in C has following syntax.

```
continue;
```

When the continue statement is encountered inside a loop, the current iteration of the loop is terminated, and the control jumps back to the top of the loop to start the next iteration.

Example of continue Statement in C

This example shows the working of the continue Statement in C.

- C

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

int i;

for (i = 0; i < 10; i++) {

if (i % 2 == 0) {

continue; // skip even numbers

}

printf("%d ", i);

}

return 0;

}
```

Output

```
1 3 5 7 9
```

Explanation:

In the above example, the for loop runs from 0 to 9, and when the loop variable i is even, the continue statement is executed, and the control moves to the next iteration of the loop. Thus, the even numbers are skipped, and only odd numbers are printed by the printf statement.

Conclusion

Conditional and unconditional branching statements in C are powerful tools that allow

programmers to change the flow of their programs based on specific conditions. The "if-else" statement allows programmers to execute code blocks based on logical conditions, while the "switch" statement allows programmers to select a code block based on a single value. Unconditional branching statements, such as the "goto" and "break" statements, allow programmers to jump to specific points in their code or exit loops and switch statements prematurely.

It is important to use branching statements carefully and judiciously to avoid creating code that is difficult to read or debug. In general, branching statements in C should be used sparingly and only when necessary to improve the clarity and organization of the code.

Branching Statements in C – FAQs

Here are some Frequently Asked Questions on Branching Statements in C.

Ques 1. Can the "switch" statement be nested in C?
Ans. No, we cannot nest the switch statement in C. This means that a switch statement in C cannot contain another switch statement inside it.

Ques 2. How many "case" statements can a "switch" statement have in C?
Ans. A "switch" statement in C can have any number of "case" statements.

Ques 3. What happens if none of the "case" statements match in a "switch" statement in C?
Ans. If none of the "case" statements match in a "switch" statement in C, the code within the "default" block is executed.

Ques 4. Can the "break" statement be used outside of loops and switch statements in C?
Ans. No, the "break" statement can only be used within loops and switch statements in C.

Ques 5. Can the "goto" statement be used to jump to a label defined in another function in C?
Ans. No, the "goto" statement can only jump to a label defined within the same function in C.

Loops in programming are used to repeat a block of code until the specified condition is met. A loop statement allows programmers to execute a statement or group of statements multiple times without repetition of code.

- C Loops in programming are used to repeat a block of code until the specified condition is met. A loop statement allows programmers to execute a statement or group of statements multiple times without repetition of code

```c
// C program to illustrate need of loops
#include <stdio.h>


int main()
{
    printf( "Hello World\n");
    printf( "Hello World\n");
    printf( "Hello World\n");
    printf( "Hello World\n");
    printf( "Hello World\n");
    printf( "Hello World\n");
    printf( "Hello World\n");
    printf( "Hello World\n");
    printf( "Hello World\n");
    printf( "Hello World\n");


    return 0;
}
```

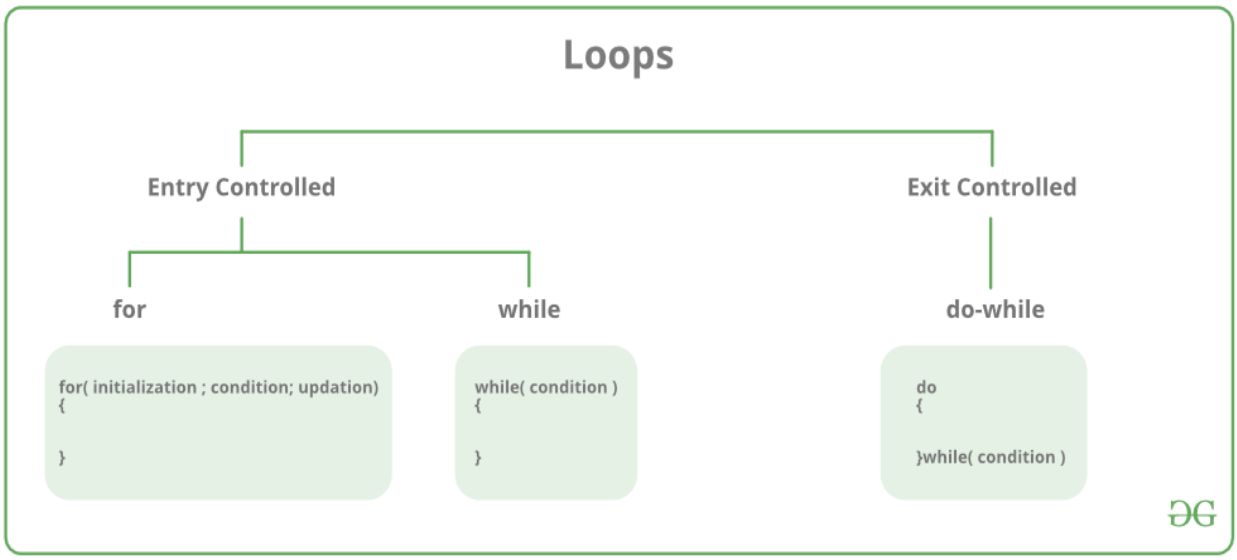
Output

```
Hello World
Hello World
Hello World
Hello World
```

```
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
```

There are mainly two types of loops in C Programming:

1. **Entry Controlled loops:** In Entry controlled loops the test condition is checked before entering the main body of the loop. **For Loop and While Loop** is Entry-controlled loops.
2. **Exit Controlled loops:** In Exit controlled loops the test condition is evaluated at the end of the loop body. The loop body will execute at least once, irrespective of whether the condition is true or false. **do-while Loop** is Exit Controlled loop.



Loop Type	Description
for loop	first Initializes, then condition check, then executes the body and at last, the update is done.
while loop	first Initializes, then condition checks, and then executes the body, and updating can be inside the body.
do-while loop	do-while first executes the body and then the condition check is done.

for Loop

for loop in C programming is a repetition control structure that allows programmers to write a loop that will be executed a specific number of times. for loop enables programmers to perform n number of steps together in a single line.

Syntax:

```
for (initialize expression; test expression; update expression)
{
    //
    // body of for loop
    //
}
```

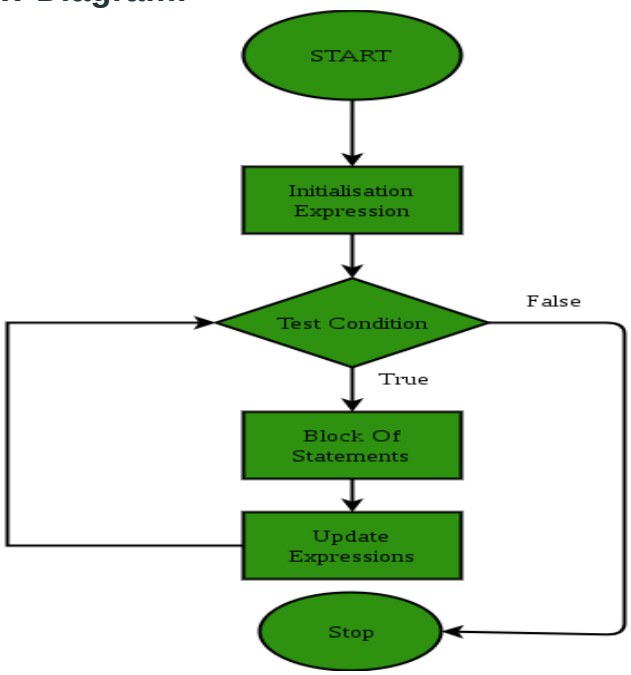
Example:

```
for(int i = 0; i < n; ++i)
{
    printf("Body of for loop which will execute till n");
}
```

In for loop, a loop variable is used to control the loop. Firstly we initialize the loop variable with some value, then check its test condition. If the statement is true then control will move to the body and the body of for loop will be executed. Steps will be repeated till the exit condition becomes true. If the test condition will be false then it will stop.

- **Initialization Expression:** In this expression, we assign a loop variable or loop counter to some value. for example: int i=1;
- **Test Expression:** In this expression, test conditions are performed. If the condition evaluates to true then the loop body will be executed and then an update of the loop variable is done. If the test expression becomes false then the control will exit from the loop. for example, i<=9;
- **Update Expression:** After execution of the loop body loop variable is updated by some value it could be incremented, decremented, multiplied, or divided by any value.

for loop Equivalent Flow Diagram:



Example:

- C

```c
// C program to illustrate for loop
#include <stdio.h>


// Driver code
int main()
{
  int i = 0;


  for (i = 1; i <= 10; i++)
  {
    printf( "Hello World\n");
  }
  return 0;
}
```

Output

```
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
```

While Loop

While loop does not depend upon the number of iterations. In for loop the number of iterations was previously known to us but in the While loop, the execution is terminated on the basis of the test condition. If the test condition will become false then it will break from the while loop else body will be executed.

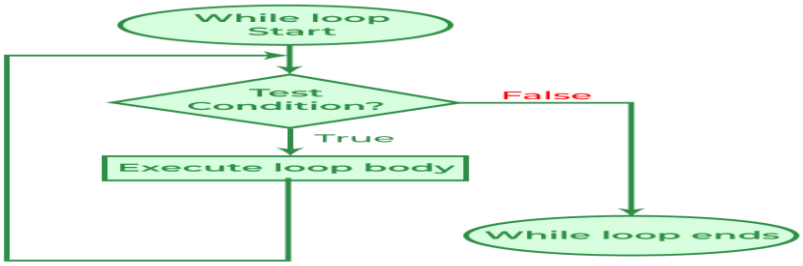
Syntax:

```
initialization_expression;


while (test_expression)
{
    // body of the while loop


    update_expression;
}
```

Flow Diagram for while loop:



- C

```
// C program to illustrate

// while loop

#include <stdio.h>



// Driver code

int main()

{

  // Initialization expression
```

```
  int i = 2;



  // Test expression

  while(i < 10)

  {

    // loop body

    printf( "Hello World\n");



    // update expression

    i++;

  }



  return 0;

}
```

Output

```
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
```

do-while Loop

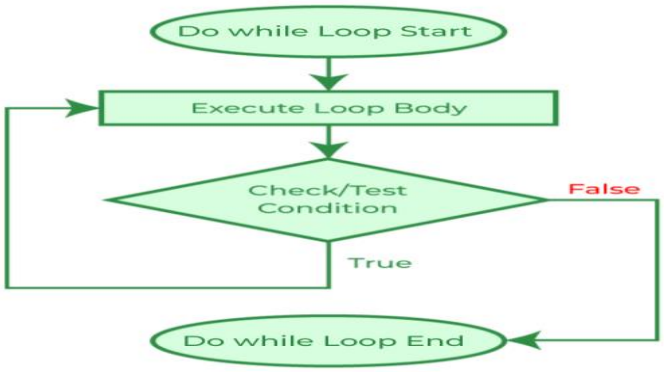
The do-while loop is similar to a while loop but the only difference lies in the do-while loop test condition which is tested at the end of the body. In the do-while loop, the loop body will **execute at least once** irrespective of the test condition.

Syntax:

```
initialization_expression;
do
{
    // body of do-while loop


    update_expression;

} while (test_expression);
```



- C

```
// C program to illustrate
// do-while loop
#include <stdio.h>


// Driver code
int main()
{
  // Initialization expression
```

```
  int i = 2;



  do

  {

    // loop body

    printf( "Hello World\n");



    // Update expression

    i++;



    // Test expression

  }  while (i < 1);



  return 0;

}
```

Output

```
Hello World
```

Above program will evaluate (i<1) as false since i = 2. But still, as it is a do-while loop the body will be executed once.

Loop Control Statements

Loop control statements in C programming are used to change execution from its normal sequence.

Name	Description
break	the break statement is used to terminate the switch and loop statement. It transfers

Name	Description
statement	the execution to the statement immediately following the loop or switch.
continue statement	continue statement skips the remainder body and immediately resets its condition before reiterating it.
goto statement	goto statement transfers the control to the labeled statement.

Infinite Loop

An infinite loop is executed when the test expression never becomes false and the body of the loop is executed repeatedly. A program is stuck in an Infinite loop when the condition is always true. Mostly this is an error that can be resolved by using Loop Control statements.

Using for loop:

- C

```
// C program to demonstrate infinite

// loops using for loop

#include <stdio.h>



// Driver code

int main ()

{

  int i;



  // This is an infinite for loop

  // as the condition expression
```

```
  // is blank

  for ( ; ; )

  {

    printf("This loop will run forever.\n");

  }



  return 0;

}
```

Output

```
This loop will run forever.

This loop will run forever.

This loop will run forever.

...
```

Using While loop:

- C

```
// C program to demonstrate

// infinite loop using while

// loop

#include <stdio.h>



// Driver code

int main()

{

  while (1)
```

```
    printf("This loop will run forever.\n");

  return 0;

}
```

Output

```
This loop will run forever.

This loop will run forever.

This loop will run forever.

...
```

Using the do-while loop:

- C

```
// C program to demonstrate

// infinite loop using do-while

// loop

#include <stdio.h>



// Driver code

int main()

{

  do

  {

    printf("This loop will run forever.\n");

  } while (1);



  return 0;
```

```
}
```

Output

```
This loop will run forever.

This loop will run forever.

This loop will run forever.
```

Chaptyer-3 Pointer and File

3.1 Pointer: introduction, Pointer and array,. Pointer and structure , Poiinter and Function

Inroduction of Pointer:-

A pointer is an address in the memory. One of the unique advantages of using C is that it provides direct access to a memory location through its address. A variable declared as int x has the address given by &x. & is a unary operator that allows the programmer to access the address of a single variable declared.

C Pointers and Array:-

Pointers & Arrays

You can also use pointers to access arrays.

Consider the following array of integers:

Example

```
int myNumbers[4] = {25, 50, 75, 100};
```

You learned from the arrays chapter that you can loop through the array elements with a `for` loop:

Example

```
int myNumbers[4] = {25, 50, 75, 100};
int i;

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

Result:

```
25
50
```

```
75
100
```

Try it Yourself »

Instead of printing the value of each array element, let's print the memory address of each array element:

Example

```
int myNumbers[4] = {25, 50, 75, 100};
int i;

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

Result:

```
0x7ffe70f9d8f0
0x7ffe70f9d8f4
0x7ffe70f9d8f8
0x7ffe70f9d8fc
```

Try it Yourself »

Note that the last number of each of the elements' memory address is different, with an addition of 4.

It is because the size of an `int` type is typically 4 bytes, remember:

Example

```
// Create an int variable
int myInt;

// Get the memory size of an int
printf("%lu", sizeof(myInt));
```

Result:

```
4
```

Try it Yourself »

So from the "memory address example" above, you can see that the compiler reserves 4 bytes of memory for each array element, which means that the entire array takes up 16 bytes (4 * 4) of memory storage:

Example

```
int myNumbers[4] = {25, 50, 75, 100};

// Get the size of the myNumbers array
printf("%lu", sizeof(myNumbers));
```

Result:

```
16
```

Try it Yourself »

How Are Pointers Related to Arrays

Ok, so what's the relationship between pointers and arrays? Well, in C, the **name of an array**, is actually a **pointer** to the **first element** of the array.

Confused? Let's try to understand this better, and use our "memory address example" above again.

The **memory address** of the **first element** is the same as the **name of the array**:

Example

```
int myNumbers[4] = {25, 50, 75, 100};

// Get the memory address of the myNumbers array
printf("%p\n", myNumbers);

// Get the memory address of the first array element
printf("%p\n", &myNumbers[0]);
```

Result:

```
0x7ffe70f9d8f0
0x7ffe70f9d8f0
```

Try it Yourself »

This basically means that we can work with arrays through pointers!

How? Since myNumbers is a pointer to the first element in myNumbers, you can use the * operator to access it:

Example

```
int myNumbers[4] = {25, 50, 75, 100};

// Get the value of the first element in myNumbers
printf("%d", *myNumbers);
```

Result:

```
25
```

Try it Yourself »

To access the rest of the elements in myNumbers, you can increment the pointer/array (+1, +2, etc):

Example

```
int myNumbers[4] = {25, 50, 75, 100};

// Get the value of the second element in myNumbers
printf("%d\n", *(myNumbers + 1));

// Get the value of the third element in myNumbers
printf("%d", *(myNumbers + 2));

// and so on..
```

Result:

```
50
75
```

Try it Yourself »

Or loop through it:

Example

```
int myNumbers[4] = {25, 50, 75, 100};
int *ptr = myNumbers;
int i;

for (i = 0; i < 4; i++) {
  printf("%d\n", *(ptr + i));
}
```

Result:

```
25
50
75
100
```

Try it Yourself »

It is also possible to change the value of array elements with pointers:

Example

```
int myNumbers[4] = {25, 50, 75, 100};

// Change the value of the first element to 13
*myNumbers = 13;

// Change the value of the second element to 17
*(myNumbers +1) = 17;

// Get the value of the first element
printf("%d\n", *myNumbers);

// Get the value of the second element
printf("%d\n", *(myNumbers + 1));
```

Result:

```
13
17
```

Structure Pointer in C

A structure pointer is defined as the pointer which points to the address of the memory block that stores a structure known as the structure pointer. Complex data structures like Linked lists, trees, graphs, etc. are created with the help of structure pointers. The structure pointer tells the address of a structure in memory by pointing the variable to the structure variable.

Example:

- C

```
// C program to demonstrate structure pointer

#include <stdio.h>



struct point {

    int value;

};



int main()

{



    struct point s;



    // Initialization of the structure pointer

    struct point* ptr = &s;
```

```
    return 0;

}
```

*In the above code **s** is an instance of struct point and **ptr** is the struct pointer because it is storing the address of struct point.*

Accessing the Structure Member with the Help of Pointers

There are two ways to access the members of the structure with the help of a structure pointer:

1. With the help of (*) asterisk or indirection operator and (.) dot operator.
2. With the help of (->) Arrow operator.

Below is the program to access the structure members using the structure pointer with the help of the dot operator.

- C

```
// C Program to demonstrate Structure pointer

#include <stdio.h>

#include <string.h>



struct Student {

    int roll_no;

    char name[30];

    char branch[40];

    int batch;

};
```

```
int main()

{



    struct Student s1;

    struct Student* ptr = &s1;



    s1.roll_no = 27;

    strcpy(s1.name, "Kamlesh Joshi");

    strcpy(s1.branch, "Computer Science And Engineering");

    s1.batch = 2019;



    printf("Roll Number: %d\n", (*ptr).roll_no);

    printf("Name: %s\n", (*ptr).name);

    printf("Branch: %s\n", (*ptr).branch);

    printf("Batch: %d", (*ptr).batch);



    return 0;

}
```

Output:

1

Below is the program to access the structure members using the structure pointer with the help of the Arrow operator. In this program, we have created a Structure Student containing structure variable s. The Structure Student has roll_no, name, branch, and batch.

- C

```c
// C Program to demonstrate Structure pointer

#include <stdio.h>

#include <string.h>



// Creating Structure Student

struct Student {

    int roll_no;

    char name[30];

    char branch[40];

    int batch;

};



// variable of structure with pointer defined

struct Student s, *ptr;



int main()

{
```

```
ptr = &s;

// Taking inputs

printf("Enter the Roll Number of Student\n");

scanf("%d", &ptr->roll_no);

printf("Enter Name of Student\n");

scanf("%s", &ptr->name);

printf("Enter Branch of Student\n");

scanf("%s", &ptr->branch);

printf("Enter batch of Student\n");

scanf("%d", &ptr->batch);



// Displaying details of the student

printf("\nStudent details are: \n");



printf("Roll No: %d\n", ptr->roll_no);

printf("Name: %s\n", ptr->name);

printf("Branch: %s\n", ptr->branch);

printf("Batch: %d\n", ptr->batch);
```

```
    return 0;

}
```

Output:

```
Enter the Roll Number of Student

27

Enter Name of Student

Kamlesh_Joshi

Enter Branch of Student

Computer_Science_And_Engineering

Enter batch of Student

2019

Student details are:

Roll No: 27

Name: Kamlesh_Joshi

Branch: Computer_Science_And_Engineering

Batch: 2019
```

Function Pointer in C

In C, like normal data pointers (int *, char *, etc), we can have pointers to functions. Following is a simple example that shows declaration and function call using function pointer.

```
#include <stdio.h>

// A normal function with an int parameter

// and void return type

void fun(int a)

{

    printf("Value of a is %d\n", a);
```

```
}



int main()

{

    // fun_ptr is a pointer to function fun()

    void (*fun_ptr)(int) = &fun;



    /* The above line is equivalent of following two

       void (*fun_ptr)(int);

       fun_ptr = &fun;

    */



    // Invoking fun() using fun_ptr

    (*fun_ptr)(10);



    return 0;

}
```

Output:

```
Value of a is 10
```

Why do we need an extra bracket around function pointers like fun_ptr in above example?
If we remove bracket, then the expression “void (*fun_ptr)(int)” becomes “void *fun_ptr(int)” which is declaration of a function that returns void pointer. See

following post for details.
How to declare a pointer to a function?
Following are some interesting facts about function pointers.

1) Unlike normal pointers, a function pointer points to code, not data. Typically a function pointer stores the start of executable code.
2) Unlike normal pointers, we do not allocate de-allocate memory using function pointers.

3) A function's name can also be used to get functions' address. For example, in the below program, we have removed address operator '&' in assignment. We have also changed function call by removing *, the program still works.

```
#include <stdio.h>

// A normal function with an int parameter

// and void return type

void fun(int a)

{

    printf("Value of a is %d\n", a);

}



int main()

{

    void (*fun_ptr)(int) = fun;  // & removed



    fun_ptr(10);  // * removed
```

```
    return 0;

}
```

Output:

```
Value of a is 10
```

4) Like normal pointers, we can have an array of function pointers. Below example in point 5 shows syntax for array of pointers.

5) Function pointer can be used in place of switch case. For example, in below program, user is asked for a choice between 0 and 2 to do different tasks.

```
#include <stdio.h>

void add(int a, int b)

{

    printf("Addition is %d\n", a+b);

}

void subtract(int a, int b)

{

    printf("Subtraction is %d\n", a-b);

}

void multiply(int a, int b)

{

    printf("Multiplication is %d\n", a*b);

}
```

```
int main()

{

    // fun_ptr_arr is an array of function pointers

    void (*fun_ptr_arr[])(int, int) = {add, subtract, multiply};

    unsigned int ch, a = 15, b = 10;



    printf("Enter Choice: 0 for add, 1 for subtract and 2 "

           "for multiply\n");

    scanf("%d", &ch);



    if (ch > 2) return 0;



    (*fun_ptr_arr[ch])(a, b);



    return 0;

}
```

```
Enter Choice: 0 for add, 1 for subtract and 2 for multiply

2

Multiplication is 150
```

6) Like normal data pointers, a function pointer can be passed as an argument and can also be returned from a function.

For example, consider the following C program where wrapper() receives a void fun() as parameter and calls the passed function.

```
// A simple C program to show function pointers as parameter
#include <stdio.h>

// Two simple functions
void fun1() { printf("Fun1\n"); }
void fun2() { printf("Fun2\n"); }

// A function that receives a simple function
// as parameter and calls the function
void wrapper(void (*fun)())
{
    fun();
}

int main()
{
    wrapper(fun1);
    wrapper(fun2);
    return 0;
```

```
}
```

This point in particular is very useful in C. In C, we can use function pointers to avoid code redundancy. For example a simple qsort() function can be used to sort arrays in ascending order or descending or by any other order in case of array of structures. Not only this, with function pointers and void pointers, it is possible to use qsort for any data type.

```
// An example for qsort and comparator

#include <stdio.h>

#include <stdlib.h>



// A sample comparator function that is used

// for sorting an integer array in ascending order.

// To sort any array for any other data type and/or

// criteria, all we need to do is write more compare

// functions.  And we can use the same qsort()

int compare (const void * a, const void * b)

{

  return ( *(int*)a - *(int*)b );

}



int main ()

{
```

```
  int arr[] = {10, 5, 15, 12, 90, 80};
  int n = sizeof(arr)/sizeof(arr[0]), i;

  qsort (arr, n, sizeof(int), compare);

  for (i=0; i<n; i++)
     printf ("%d ", arr[i]);
  return 0;
}
```

Output:

```
5 10 12 15 80 90
```

Similar to qsort(), we can write our own functions that can be used for any data type and can do different tasks without code redundancy. Below is an example search function that can be used for any data type. In fact we can use this search function to find close elements (below a threshold) by writing a customized compare function.

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

// A compare function that is used for searching an integer
// array
bool compare (const void * a, const void * b)
{
```

```
  return ( *(int*)a == *(int*)b );
}


// General purpose search() function that can be used
// for searching an element *x in an array arr[] of
// arr_size. Note that void pointers are used so that
// the function can be called by passing a pointer of
// any type.  ele_size is size of an array element
int search(void *arr, int arr_size, int ele_size, void *x,
           bool compare (const void * , const void *))
{
    // Since char takes one byte, we can use char pointer
    // for any type/ To get pointer arithmetic correct,
    // we need to multiply index with size of an array
    // element ele_size
    char *ptr = (char *)arr;


    int i;
    for (i=0; i<arr_size; i++)
        if (compare(ptr + i*ele_size, x))
           return i;
```

```
    // If element not found
    return -1;
}


int main()
{
    int arr[] = {2, 5, 7, 90, 70};
    int n = sizeof(arr)/sizeof(arr[0]);
    int x = 7;
    printf ("Returned index is %d ", search(arr, n,
                                sizeof(int), &x, compare));
    return 0;
}
```

Output:

```
Returned index is 2
```

The above search function can be used for any data type by writing a separate customized compare().

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()

C fprintf()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:

```c
#include <stdio.h>
main(){
  FILE *fp;
  fp = fopen("file.txt", "w");//opening file
  fprintf(fp, "Hello file by fprintf...\n");//writing data into file
  fclose(fp);//closing file
}
```

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:

```c
int fscanf(FILE *stream, const char *format [, argument, ...])
```

Example:

```c
#include <stdio.h>
main(){
  FILE *fp;
  char buff[255];//creating char array to store data of file
  fp = fopen("file.txt", "r");
  while(fscanf(fp, "%s", buff)!=EOF){
  printf("%s ", buff );
  }
  fclose(fp);
}
```

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.

```c
#include <stdio.h>
void main()
{
    FILE *fptr;
    int id;
    char name[30];
    float salary;
    fptr = fopen("emp.txt", "w+");/*  open for writing */
    if (fptr == NULL)
    {
        printf("File does not exists \n");
        return;
    }
    printf("Enter the id\n");
    scanf("%d", &id);
    fprintf(fptr, "Id= %d\n", id);
    printf("Enter the name \n");
    scanf("%s", name);
    fprintf(fptr, "Name= %s\n", name);
    printf("Enter the salary\n");
    scanf("%f", &salary);
    fprintf(fptr, "Salary= %.2f\n", salary);
    fclose(fptr);
}
```

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
```

and fscanf() example

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:

```
int fputc(int c, FILE *stream)
```

Example:

```
#include <stdio.h>
main(){
  FILE *fp;
  fp = fopen("file1.txt", "w");//opening file
  fputc('a',fp);//writing single character into file
  fclose(fp);//closing file
}
```

file1.txt

Backward Skip 10sPlay VideoForward Skip 10s

```
a
```

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:

```
int fgetc(FILE *stream)
```

Example:

```
#include<stdio.h>
#include<conio.h>
void main(){
FILE *fp;
char c;
clrscr();
fp=fopen("myfile.txt","r");

while((c=fgetc(fp))!=EOF){
printf("%c",c);
}
fclose(fp);
getch();
}
```

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 fgets() functions.

Writing File : fputs() function

The fputs() function writes a line of characters into file. It outputs string to a stream.

Syntax:

```
int fputs(const char *s, FILE *stream)
```

Example:

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

```
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:

```
#include <stdio.h>
void main(){
  FILE *fp;

  fp = fopen("myfile.txt","w+");
  fputs("This is javatpoint", fp);

  fseek( fp, 7, SEEK_SET );
  fputs("sonoo jaiswal", fp);
  fclose(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:

```
int fprintf(FILE *stream, const char *format [, argument, ...])
```

Example:

```
#include <stdio.h>
main(){
  FILE *fp;
  fp = fopen("file.txt", "w");//opening file
  fprintf(fp, "Hello file by fprintf...\n");//writing data into file
  fclose(fp);//closing file
}
```

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.

Backward Skip 10sPlay VideoForward Skip 10s

Syntax:

```
int fscanf(FILE *stream, const char *format [, argument, ...])
```

Example:

```
#include <stdio.h>
main(){
  FILE *fp;
  char buff[255];//creating char array to store data of file
  fp = fopen("file.txt", "r");
  while(fscanf(fp, "%s", buff)!=EOF){
  printf("%s ", buff );
  }
  fclose(fp);
}
```

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(fptr, "Id= %d\n", id);
17.    printf("Enter the name \n");
18.    scanf("%s", name);
19.    fprintf(fptr, "Name= %s\n", name);
20.    printf("Enter the salary\n");
21.    scanf("%f", &salary);
22.    fprintf(fptr, "Salary= %.2f\n", salary);
23.    fclose(fptr);
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 direC 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:

```
int fputc(int c, FILE *stream)
```

Example:

```
#include <stdio.h>
main(){
  FILE *fp;
  fp = fopen("file1.txt", "w");//opening file
  fputc('a',fp);//writing single character into file
  fclose(fp);//closing file
}
```

file1.txt

```
a
```

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:

```
int fgetc(FILE *stream)
```

Example:

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

ctory, you will see emp.txt file. It will have following information.

emp.txt

```
Id= 1
Name= sonoo
Salary= 120000
```