Івано-Франківський національний технічний університет нафти і газу

1
1
1
1
1
1
1
1
1
110
1
1
120
1
1
#include <stdio.h>
10
1
1
1
1
140
1

Р.І. Храбатин
Л.В. Саманів

ОСНОВИ ПРОГРАМУВАННЯ

2
2
2
2
2
2
2
2
2
110
2
2
120
2
2
#include <stdio.h>
20
2
2
2
2
140
2

3
3
3
3
3
3
3
3
3
110
3
3
120
3
3
#include <stdio.h>
30
3
3
3
3
140
3

4
4
4
4
4
4
4
4
4
110
4
4
120
4
4
#include <stdio.h>
40
4
4
4
4
140
4

2020

5
5
5
5
5
5
5
5
5
110
5
5
120
5
5
#include <stdio.h>
50
5
5
5
5
140
5

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Івано-Франківський національний технічний університет нафти і газу

6
6
6
6
6
6
6
6
6
110
6
6
120
6
6
#include <stdio.h>
60
6
6
6
6
140
6

Р.І. Храбатин
Л.В. Саманів

7
7
7
7
7
7
7
7
7
110
7
7
120
7
7
```
#include <stdio.h>
```
70
7
7
7
7
140
7

ОСНОВИ ПРОГРАМУВАННЯ

Для студентів освітньо-професійної програми «Інженерія програмного забезпечення» рівня вищої освіти «бакалавр»

8
8
8
8
8
8
8
8
8
110
8
8
120
8
8
#include <stdio.h>
80
8
8
8
8
140
8

Рекомендовано методичною радою університету

9
9
9
9
9
9
9
9
9
110
9
9
120
9
9
```
#include <stdio.h>
```
90
9
9
9
9
140
9

Івано-Франківськ
2020

```
10
10
10
10
10
10
10
10
10
110
10
10
120
10
10
#include <stdio.h>
100
10
10
10
10
140
10
```

Ivano-Frankivsk National Technical University of Oil and Gas

C programming language

11
11
11
11
11
11
11
11
11
110
11
11
120
11
11
```
#include <stdio.h>
```
110
11
11
11
11
140
11

R. Khrabatyn
L. Samaniv

12
12
12
12
12
12
12
12
12
110
12
12
120
12
12
#include <stdio.h>
120
12
12
12
12
140
12

Ivano-Frankivsk

13
13
13
13
13
13
13
13
13
110
13
13
120
13
13
#include <stdio.h>
130
13
13
13
13
140
13

2020

14
14
14
14
14
14
14
14
14
110
14
14
120
14
14
```
#include <stdio.h>
```
140
14
14
14
14
140
14

PROGRAM STRUCTURE 6
Hello World Example **6**
Compile and Execute C Program **7**
1. BASIC SYNTAX
Tokens in C **8**
Semicolons **8**
Comments **8**
Identifiers **9**
Keywords **9**
Whitespace in C **10**
2. DATA TYPES

15
15
15
15
15
15
15
15
15
110
15
15
120
15
15
#include <stdio.h>
150
15
15
15
15
140
15

Integer Types **11**
Floating-Point Types **13**
The void Type **14**
3. VARIABLES
Variable Definition in C **15**
Variable Declaration in C **16**
Lvalues and Rvalues in C **18**
4. CONSTANTS AND LITERALS
Integer Literals **19**
Floating-point Literals **20**
Character Constants **20**

16
16
16
16
16
16
16
16
16
110
16
16
120
16
16
#include <stdio.h>
160
16
16
16
16
140
16

String Literals **21**
Defining Constants **22**
The #define Preprocessor 22
The const Keyword 23
5. STORAGE CLASSES
The auto Storage Class **24**
The register Storage Class **24**
The static Storage Class **25**
The extern Storage Class **26**
6. OPERATORS
Arithmetic Operators **28**

17
17
17
17
17
17
17
17
17
110
17
17
120
17
17
#include <stdio.h>
170
17
17
17
17
140
17

Relational Operators **30**
Logical Operators **32**
Bitwise Operators **34**
Assignment Operators **37**
Misc Operators ↦ sizeof & ternary **40**
Operators Precedence in C **41**
7. DECISION MAKING
if Statement **46**
if…else Statement **48**
if...else if...else Statement **49**
Nested if Statements **51**

18
18
18
18
18
18
18
18
18
110
18
18
120
18
18

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

180
18
18
18
18
140
18

switch Statement **53**
Nested switch Statements **55**
The ? : Operator **57**
8. LOOPS
while Loop **59**
for Loop **61**
do…while Loop **63**
Nested Loops **65**
Loop Control Statements **67**
break Statement **68**
continue Statement **70**

19
19
19
19
19
19
19
19
19
110
19
19
120
19
19

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

190
19
19
19
19
140
19

goto Statement **72**
The Infinite Loop **74**
9. FUNCTIONS
Defining a Function **76**
Function Declarations **77**
Calling a Function **78**
Function Arguments **79**
Call by Value **80**
Call by Reference **81**
10. SCOPE RULES
Local Variables **84**

20
20
20
20
20
20
20
20
20
110
20
20
120
20
20

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

200
20
20
20
20
140
20

Global Variables **85**
Formal Parameters **86**
Initializing Local and Global Variables **87**
11. ARRAYS
Declaring Arrays **89**
Initializing Arrays **89**
Accessing Array Elements **90**
Arrays in Detail **91**
12. POINTERS
What are Pointers? **101**
How to Use Pointers? **102**

21
21
21
21
21
21
21
21
21
110
21
21
120
21
21
#include <stdio.h>
210
21
21
21
21
140
21

NULL Pointers **103**
Pointers in Detail **104**
13. STRINGS
14. STRUCTURES
Defining a Structure **120**
Accessing Structure Members **121**
Structures as Function Arguments **122**
Pointers to Structures **124**
Bit Fields **126**
15. UNIONS
Defining a Union **128**

22
22
22
22
22
22
22
22
22
110
22
22
120
22
22

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

220
22
22
22
22
140
22

Accessing Union Members **129**
16. BIT FIELDS
Bit Field Declaration **133**
17. TYPEDEF
typedef vs #define **137**
18. INPUT AND OUTPUT 139
The Standard Files **139**
The getchar() and putchar() Functions **139**
The gets() and puts() Functions **140**
The scanf() and printf() Functions **141**
19. FILE

23
23
23
23
23
23
23
23
23
110
23
23
120
23
23
#include <stdio.h>
230
23
23
23
23
140
23

Opening Files **143**
Closing a File **144**
Writing a File **144**
Reading a File **145**
Binary I/O Functions **146**

24
24
24
24
24
24
24
24
24
110
24
24
120
24
24
#include <stdio.h>
240
24
24
24
24
140
24

Programming structure

Before we study the basic building blocks of the C programming language, let us look at a bare minimum C program structure so that we can take it as a reference in the upcoming chapters.

Hello World Example

A C program basically consists of the following parts:

- Preprocessor Commands
- Functions

25
25
25
25
25
25
25
25
25
110
25
25
120
25
25

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

250
25
25
25
25
140
25

- Variables
- Statements & Expressions
- Comments

Let us look at a simple code that would print the words "Hello World":

```
#include <stdio.h>


int main()
{
   /* my first program in C */
```

26
26
26
26
26
26
26
26
26
110
26
26
120
26
26

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

260
26
26
26
26
140
26

```
    printf("Hello, World! \n");

    return 0;
 }
```

Let us take a look at the various parts of the above program:

1. The first line of the program *#include <stdio.h>* is a preprocessor command, which tells a C compiler to include stdio.h file before going to actual compilation.
2. The next line *int main()* is the main function where the program execution begins.
3. The next line /*...*/ will be ignored by the compiler and it has been put to add

27
27
27
27
27
27
27
27
27
110
27
27
120
27
27
```
#include <stdio.h>
```
270
27
27
27
27
140
27

additional comments in the program. So such lines are called comments in the program.

28
28
28
28
28
28
28
28
28
110
28
28
120
28
28
#include <stdio.h>
280
28
28
28
28
140
28

4. The next line *printf(...)* is another function available in C which causes the message "Hello, World!" to be displayed on the screen.

5. The next line **return 0;** terminates the main() function and returns the value 0.

Compile and Execute C Program

Let us see how to save the source code in a file, and how to compile and run it. Following are the simple steps:

1. Open a text editor and add the above-mentioned code.
2. Save the file as *hello.c*

29
29
29
29
29
29
29
29
29
110
29
29
120
29
29

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

290
29
29
29
29
140
29

3. Open a command prompt and go to the directory where you have saved the file.
4. Type *gcc hello.c* and press enter to compile your code.
5. If there are no errors in your code, the command prompt will take you to the next line and would generate *a.out* executable file.
6. Now, type *a.out* to execute your program.
7. You will see the output *"Hello World"* printed on the screen.

```
$ gcc hello.c

$ ./a.out Hello,

World!
```

Make sure the gcc compiler is in your path and that you are running it in the directory

30
30
30
30
30
30
30
30
30
110
30
30
120
30
30

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

300
30
30
30
30
140
30

containing the source file hello.c.

You have seen the basic structure of a C program, so it will be easy to understand other basic building blocks of the C programming language.

Tokens in C

```
 printf("Hello, World! \n");
```

A C program consists of various tokens and a token is either a keyword, an identifier, a constant, a string literal, or a symbol. For example, the following C statement consists of five tokens:The individual tokens are:

31
31
31
31
31
31
31
31
31
110
31
31
120
31
31

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

310
31
31
31
31
140
31

```
printf (
"Hello, World! \n"
)
;
```

Semicolons

In a C program, the semicolon is a statement terminator. That is, each individual statement must be ended with a semicolon. It indicates the end of one logical entity.

Given below are two different statements:

32
32
32
32
32
32
32
32
32
110
32
32
120
32
32

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

320
32
32
32
32
140
32

```
printf("Hello, World! \n");
return 0;
```

Comments

```
/* my first program in C */
```

Comments are like helping text in your C program and they are ignored by the compiler. They start with /* and terminate with the characters */ as shown below:

You cannot have comments within comments and they do not occur within a string or character literals.

33
33
33
33
33
33
33
33
33
110
33
33
120
33
33

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

330
33
33
33
33
140
33

Identifiers

A C identifier is a name used to identify a variable, function, or any other user- defined item. An identifier starts with a letter A to Z, a to z, or an underscore '_' followed by zero or more letters, underscores, and digits (0 to 9).

C does not allow punctuation characters such as @, $, and % within identifiers. C is a **case-sensitive** programming language. Thus, *Manpower* and *manpower* are two different identifiers in C. Here are some examples of acceptable identifiers:

mohd	zara	abc	move_name	a_123

34
34
34
34
34
34
34
34
34
110
34
34
120
34
34
#include <stdio.h>
340
34
34
34
34
140
34

myname50	temp	j	a23b9	retVal

Keywords

The following list shows the reserved words in C. These reserved words may not be used as constants or variables or any other identifier names.

auto	else	long	switch
break	enum	register	typedef

35
35
35
35
35
35
35
35
35
110
35
35
120
35
35
#include <stdio.h>
350
35
35
35
35
140
35

case	extern	return	union
char	float	short	unsigned
const	for	signed	void
continue	goto	sizeof	volatile
default	if	static	while

36
36
36
36
36
36
36
36
36
110
36
36
120
36
36

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

360
36
36
36
36
140
36

do	int	struct	Packed
double			

Whitespace in C

A line containing only whitespace, possibly with a comment, is known as a blank line, and a C compiler totally ignores it.

```
int age;
```

Whitespace is the term used in C to describe blanks, tabs, newline characters and comments. Whitespace separates one part of a statement from another and enables the compiler to

37
37
37
37
37
37
37
37
37
110
37
37
120
37
37

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

370
37
37
37
37
140
37

identify where one element in a statement, such as int, ends and the next element begins. Therefore, in the following statement:there must be at least one whitespace character (usually a space) between int and age for the compiler to be able to distinguish them. On the other hand, in the following statement:

```
fruit = apples + oranges;   // get the total fruit
```

no whitespace characters are necessary between fruit and =, or between = and apples, although you are free to include some if you wish to increase readability.

Data types in C refer to an extensive system used for declaring variables or functions of different types. The type of a variable determines how much space it occupies in storage and how the bit pattern stored is interpreted.

38
38
38
38
38
38
38
38
38
110
38
38
120
38
38

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

380
38
38
38
38
140
38

The types in C can be classified as follows:

.N.	Types and Description
	Basic Types: They are arithmetic types and are further classified into: (a) integer types and (b) floating-point types.
	Enumerated types:

39
39
39
39
39
39
39
39
39
110
39
39
120
39
39

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

390
39
39
39
39
140
39

	They are again arithmetic types and they are used to define variables that can only assign certain discrete integer values throughout the program.
	The type void: The type specifier *void* indicates that no value is available.
	Derived types: They include (a) Pointer types, (b) Array types, (c) Structure types, (d) Union types, and (e) Function types.

The array types and structure types are referred collectively as the aggregate types. The

40
40
40
40
40
40
40
40
40
110
40
40
120
40
40
#include <stdio.h>
400
40
40
40
40
140
40

type of a function specifies the type of the function's return value. We will see the basic types in the following section, whereas other types will be covered in the upcoming chapters.

Integer Types

The following table provides the details of standard integer types with their storage sizes and value ranges:

Type	Storage	Value range

41
41
41
41
41
41
41
41
41
110
41
41
120
41
41
#include <stdio.h>
410
41
41
41
41
140
41

	size	
char	1 byte	-128 to 127 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127
int	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647

42
42
42
42
42
42
42
42
42
110
42
42
120
42
42

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

420
42
42
42
42
140
42

unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
long	4 bytes	-2,147,483,648 to 2,147,483,647
unsigned long	4 bytes	0 to 4,294,967,295

43
43
43
43
43
43
43
43
43
110
43
43
120
43
43

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

430
43
43
43
43
140
43

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

int main()
{
   printf("Storage size for int : %d \n", sizeof(int));
```

To get the exact size of a type or a variable on a particular platform, you can use the **sizeof** operator. The expressions *sizeof(type)* yields the storage size of the object or type in bytes. Given below is an example to get the size of int type on any machine:

44
44
44
44
44
44
44
44
44
110
44
44
120
44
44

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

440
44
44
44
44
140
44

```
    return 0;
}
Storage size for int : 4
```

When you compile and execute the above program, it produces the following result on Linux:

Floating-Point Types

The following table provides the details of standard floating-point types with storage sizes and value ranges and their precision:

45
45
45
45
45
45
45
45
45
110
45
45
120
45
45

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

450
45
45
45
45
140
45

Type	Storage size	Value range	Precision
float	4 byte	1.2E-38 to 3.4E+38	6 decimal places
double	8 byte	2.3E-308 to 1.7E+308	15 decimal places
long double	10 byte	3.4E-4932 to 1.1E+4932	19 decimal places

46
46
46
46
46
46
46
46
46
110
46
46
120
46
46
#include <stdio.h>
460
46
46
46
46
140
46

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

int main()
{
   printf("Storage size for float : %d \n", sizeof(float));
   printf("Minimum float positive value: %E\n", FLT_MIN );
   printf("Maximum float positive value: %E\n", FLT_MAX );
```

47
47
47
47
47
47
47
47
47
110
47
47
120
47
47

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

470
47
47
47
47
140
47

```
    printf("Precision value: %d\n", FLT_DIG );


    return 0;
```

The header file float.h defines macros that allow you to use these values and other details about the binary representation of real numbers in your programs. The following example prints the storage space taken by a float type and its range values:

```
 }
 Storage size for float : 4


48
48
48
48
48
48
48
48
48
110
48
48
120
48
48
#include <stdio.h>
480
48
48
48
48
140
48
```

```
Minimum float positive value: 1.175494E-38
Maximum float positive value: 3.402823E+38
Precision value: 6
```

When you compile and execute the above program, it produces the following result on Linux:

The void Type

The void type specifies that no value is available. It is used in three kinds of situations:

49
49
49
49
49
49
49
49
49
110
49
49
120
49
49

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

490
49
49
49
49
140
49

S.N.	Types and Description
1	**Function returns as void** There are various functions in C which do not return any value or you can say they return void. A function with no return value has the return type as void. For example, **void exit (int status);**
2	**Function arguments as void** There are various functions in C which do not accept any parameter. A function with no parameter can accept a void. For example, **int rand(void);**

50
50
50
50
50
50
50
50
50
110
50
50
120
50
50

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

500
50
50
50
50
140
50

3	**Pointers to void** A pointer of type void * represents the address of an object, but not its type. For example, a memory allocation function **void *malloc(size_t size);** returns a pointer to void which can be casted to any data type.

51
51
51
51
51
51
51
51
51
110
51
51
120
51
51
#include <stdio.h>
510
51
51
51
51
140
51

Variables

A variable is nothing but a name given to a storage area that our programs can manipulate. Each variable in C has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.

The name of a variable can be composed of letters, digits, and the underscore character. It must begin with either a letter or an underscore. Upper and lowercase letters are distinct because C is case-sensitive. Based on the basic types explained in the previous chapter, there will be the following basic variable types:

52
52
52
52
52
52
52
52
52
110
52
52
120
52
52
#include <stdio.h>
520
52
52
52
52
140
52

Type	Description
char	Typically a single octet (one byte). This is an integer type.
int	The most natural size of integer for the machine.
float	A single-precision floating point value.
double	A double-precision floating point value.

53
53
53
53
53
53
53
53
53
110
53
53
120
53
53

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

530
53
53
53
53
140
53

void	Represents the absence of type.

C programming language also allows to define various other types of variables, which we will cover in subsequent chapters like Enumeration, Pointer, Array, Structure, Union, etc. For this chapter, let us study only basic variable types.

Variable Definition in C

```
type variable_list;
```

A variable definition tells the compiler where and how much storage to create for the variable. A variable definition specifies a data type and contains a list of one or more variables of

54
54
54
54
54
54
54
54
54
110
54
54
120
54
54

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

540
54
54
54
54
140
54

that type as follows:

```
55
55
55
55
55
55
55
55
55
110
55
55
120
55
55
#include <stdio.h>
550
55
55
55
55
140
55
```

```
int    i, j, k;
char   c, ch; float
       f, salary;
double d;
```

Here, **type** must be a valid C data type including char, w_char, int, float, double, bool, or any user-defined object; and **variable_list** may consist of one or more identifier names separated by commas. Some valid declarations are shown here:The line **int i, j, k;** declares and defines the variables i, j and k; which instruct the compiler to create variables named i, j, and k of type int.

```
type variable_name = value;
```

56
56
56
56
56
56
56
56
56
110
56
56
120
56
56

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

560
56
56
56
56
140
56

Variables can be initialized (assigned an initial value) in their declaration. The initializer consists of an equal sign followed by a constant expression as follows:Some examples are:

```
// declaration of d and f.

// definition and initializing d and f.

// definition and initializes z.

// the variable x has the value 'x'.
extern int d = 3, f = 5;

int d = 3, f = 5; byte z = 22;

char x = 'x';
```

57
57
57
57
57
57
57
57
57
110
57
57
120
57
57

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

570
57
57
57
57
140
57

For definition without an initializer: variables with static storage duration are implicitly initialized with NULL (all bytes have the value 0); the initial value of all other variables are undefined.

Variable Declaration in C

A variable declaration provides assurance to the compiler that there exists a variable with the given type and name so that the compiler can proceed for further compilation without requiring the complete detail about the variable. A variable declaration has its meaning at the time of compilation only, the compiler needs actual variable declaration at the time of linking the program.

58
58
58
58
58
58
58
58
58
110
58
58
120
58
58

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

580
58
58
58
58
140
58

A variable declaration is useful when you are using multiple files and you define your variable in one of the files which will be available at the time of linking the program. You will use the keyword **extern** to declare a variable at any place. Though you can declare a variable multiple times in your C program, it can be defined only once in a file, a function, or a block of code.

Example

Try the following example, where variables have been declared at the top, but they have been defined and initialized inside the main function:

```
59
59
59
59
59
59
59
59
59
110
59
59
120
59
59
#include <stdio.h>
590
59
59
59
59
140
59
```

```
#include <stdio.h>

// Variable declaration:
extern int a, b;
extern int c;
extern float f;
```

60
60
60
60
60
60
60
60
60
110
60
60
120
60
60

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

600
60
60
60
60
140
60

```
 int main ()
 {
   /* variable definition: */
   int a, b;
   int c;
   float f;

   /* actual initialization */ a
```

61
61
61
61
61
61
61
61
61
110
61
61
120
61
61

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

610
61
61
61
61
140
61

```
   = 10;
   b = 20;

   c = a + b;
   printf("value of c : %d \n", c);

   f = 70.0/3.0;
   printf("value of f : %f \n", f);
```

62
62
62
62
62
62
62
62
62
110
62
62
120
62
62

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

620
62
62
62
62
140
62

```
   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
value of c : 30
value of f : 23.333334
```

The same concept applies on function declaration where you provide a function name at the time of its declaration and its actual definition can be given anywhere else. For example:

63
63
63
63
63
63
63
63
63
110
63
63
120
63
63

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

630
63
63
63
63
140
63

64
64
64
64
64
64
64
64
64
110
64
64
120
64
64
#include <stdio.h>
640
64
64
64
64
140
64

```
 // function declaration
 int func();

 int main()
 {
     // function call
     int i = func();
```

65
65
65
65
65
65
65
65
65
110
65
65
120
65
65

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

650
65
65
65
65
140
65

```
}

// function definition
int func()
{
    return 0;
}
```

Lvalues and Rvalues in C

66
66
66
66
66
66
66
66
66
110
66
66
120
66
66
#include <stdio.h>
660
66
66
66
66
140
66

There are two kinds of expressions in C:

· **lvalue :** Expressions that refer to a memory location are called "lvalue" expressions. An lvalue may appear as either the left-hand or right-hand side of an assignment.

· **rvalue :** The term rvalue refers to a data value that is stored at some address in memory. An rvalue is an expression that cannot have a value assigned to it which means an rvalue may appear on the right-hand side but not on the left-hand side of an assignment.

```
// valid statement

// invalid statement; would generate compile-time error
int g = 20;

10 = 20;
```

67
67
67
67
67
67
67
67
67
110
67
67
120
67
67

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

670
67
67
67
67
140
67

Variables are lvalues and so they may appear on the left-hand side of an assignment. Numeric literals are rvalues and so they may not be assigned and cannot appear on the left-hand side. Take a look at the following valid and invalid statements:

68
68
68
68
68
68
68
68
68
110
68
68
120
68
68
#include <stdio.h>
680
68
68
68
68
140
68

Constant and literals

Constants refer to fixed values that the program may not alter during its execution. These fixed values are also called **literals**.

Constants can be of any of the basic data types like *an integer constant, a floating constant, a character constant, or a string literal*. There are enumeration constants as well.

Constants are treated just like regular variables except that their values cannot be modified after their definition.

Integer Literals

```
69
69
69
69
69
69
69
69
69
110
69
69
120
69
69
#include <stdio.h>
690
69
69
69
69
140
69
```

An integer literal can be a decimal, octal, or hexadecimal constant. A prefix specifies the base or radix: 0x or 0X for hexadecimal, 0 for octal, and nothing for decimal.

An integer literal can also have a suffix that is a combination of U and L, for unsigned and long, respectively. The suffix can be uppercase or lowercase and can be in any order.

```
/* Legal */

/* Legal */

/* Legal */

/* Illegal: 8 is not an octal digit */

/* Illegal: cannot repeat a suffix */
212

70
70
70
70
70
70
70
70
70
110
70
70
120
70
70
#include <stdio.h>
700
70
70
70
70
140
70
```

```
215u

0xFeeL 078

032UU
```

Here are some examples of integer literals:Following are other examples of various types of integer literals:

```
/* decimal */

/* octal */

/* hexadecimal */

/* int */
```

71
71
71
71
71
71
71
71
71
110
71
71
120
71
71

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

710
71
71
71
71
140
71

```
/* unsigned int */

/* long */

/* unsigned long */
85

0213

0x4b 30

30u 30l

30ul
```

72
72
72
72
72
72
72
72
72
110
72
72
120
72
72

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

720
72
72
72
72
140
72

73
73
73
73
73
73
73
73
73
110
73
73
120
73
73
#include <stdio.h>
730
73
73
73
73
140
73

Floating-point Literals

A floating-point literal has an integer part, a decimal point, a fractional part, and an exponent part. You can represent floating point literals either in decimal form or exponential form.

While representing decimal form, you must include the decimal point, the exponent, or both; and while representing exponential form, you must include the integer part, the fractional part, or both. The signed exponent is introduced by e or E.

Here are some examples of floating-point literals:

```
/* Legal */

/* Legal */

/* Illegal: incomplete exponent */
```

74
74
74
74
74
74
74
74
74
110
74
74
120
74
74

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

740
74
74
74
74
140
74

```
/* Illegal: no decimal or exponent */

/* Illegal: missing integer or fraction */
3.14159

314159E-5L

510E

210f

.e55
```

Character Constants

```
75
75
75
75
75
75
75
75
75
110
75
75
120
75
75
#include <stdio.h>
750
75
75
75
75
140
75
```

Character literals are enclosed in single quotes, e.g., 'x' can be stored in a simple variable of **char** type.

A character literal can be a plain character (e.g., 'x'), an escape sequence (e.g., '\t'), or a universal character (e.g., '\u02C0').

There are certain characters in C that represent special meaning when preceded by a backslash, for example, newline (\n) or tab (\t). Here, you have a list of such escape sequence codes:

Escape	Meaning

76
76
76
76
76
76
76
76
76
110
76
76
120
76
76
#include <stdio.h>
760
76
76
76
76
140
76

sequence	
\\	\ character
\'	' character
\"	" character
\?	? character
\a	Alert or bell

77
77
77
77
77
77
77
77
77
110
77
77
120
77
77
```
#include <stdio.h>
```
770
77
77
77
77
140
77

78
78
78
78
78
78
78
78
78
110
78
78
120
78
78
#include <stdio.h>
780
78
78
78
78
140
78

\b	Backspace
\f	Form feed
\n	Newline
\r	Carriage return
\t	Horizontal tab

79
79
79
79
79
79
79
79
79
110
79
79
120
79
79

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

790
79
79
79
79
140
79

\v	Vertical tab
\ooo	Octal number of one to three digits
\xhh . . .	Hexadecimal number of one or more digits

Following is the example to show a few escape sequence characters:

```
#include <stdio.h>

80
80
80
80
80
80
80
80
80
110
80
80
120
80
80
#include <stdio.h>
800
80
80
80
80
140
80
```

```
int main()
{
   printf("Hello\tWorld\n\n");

   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Hello   World
```

81
81
81
81
81
81
81
81
81
110
81
81
120
81
81

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

810
81
81
81
81
140
81

String Literals

String literals or constants are enclosed in double quotes "". A string contains characters that are similar to character literals: plain characters, escape sequences, and universal characters.

82
82
82
82
82
82
82
82
82
110
82
82
120
82
82
#include <stdio.h>
820
82
82
82
82
140
82

You can break a long line into multiple lines using string literals and separating them using whitespaces.

```
 "hello, dear"


 "hello, \


 dear"
```

```
83
83
83
83
83
83
83
83
83
110
83
83
120
83
83
#include <stdio.h>
830
83
83
83
83
140
83
```

```
"hello, " "d" "ear"
```

Here are some examples of string literals. All the three forms are identical strings.

Defining Constants

There are two simple ways in C to define constants:

- Using **#define** preprocessor
- Using **const** keyword

The #define Preprocessor

Given below is the form to use #define preprocessor to define a constant:

```
#define identifier value
```

84
84
84
84
84
84
84
84
84
110
84
84
120
84
84

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

840
84
84
84
84
140
84

The following example explains it in detail:

```
#include <stdio.h>


#define LENGTH 10
#define WIDTH  5
#define NEWLINE '\n'

int main()
```

```
85
85
85
85
85
85
85
85
85
110
85
85
120
85
85
#include <stdio.h>
850
85
85
85
85
140
85
```

```
 {

    int area;

    area = LENGTH * WIDTH;
```

86
86
86
86
86
86
86
86
86
110
86
86
120
86
86

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

860
86
86
86
86
140
86

```
   printf("value of area : %d", area);
   printf("%c", NEWLINE);

   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
value of area : 50
```

87
87
87
87
87
87
87
87
87
110
87
87
120
87
87

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

870
87
87
87
87
140
87

The const Keyword

You can use **const** prefix to declare constants with a specific type as follows:

```
 const type variable = value;
```

The following example explains it in detail:

```
}
return 0;
area = LENGTH * WIDTH;
printf("value of area : %d", area); printf("%c", NEWLINE);

const char NEWLINE = '\n';

int area;

88
88
88
88
88
88
88
88
88
110
88
88
120
88
88
#include <stdio.h>
880
88
88
88
88
140
88
```

```
= 5;
WIDTH
LENGTH = 10;
int main()
```

```
{
   const int const int
```

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

When the above code is compiled and executed, it produces the following result:

```
 value of area : 50
```

Note that it is a good programming practice to define constants in CAPITALS.

89
89
89
89
89
89
89
89
89
110
89
89
120
89
89

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

890
89
89
89
89
140
89

90
90
90
90
90
90
90
90
90
110
90
90
120
90
90
```
#include <stdio.h>
```
900
90
90
90
90
140
90

Classes

A storage class defines the scope (visibility) and life-time of variables and/or functions within a C Program. They precede the type that they modify. We have four different storage classes in a C program:

- auto
- register
- static
- extern

The auto Storage Class

91
91
91
91
91
91
91
91
91
110
91
91
120
91
91

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

910
91
91
91
91
140
91

The **auto** storage class is the default storage class for all local variables.

```
{
   int mount;
   auto int month;
}
```

The example above defines two variables within the same storage class. ‘auto’ can only be used within functions, i.e., local variables.

The register Storage Class

92
92
92
92
92
92
92
92
92
110
92
92
120
92
92

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

920
92
92
92
92
140
92

```
{
   register int  miles;
}
```

The **register** storage class is used to define local variables that should be stored in a register instead of RAM. This means that the variable has a maximum size equal to the register size (usually one word) and can't have the unary '&' operator applied to it (as it does not have a memory location).The register should only be used for variables that require quick access such as counters. It should also be noted that defining 'register' does not mean that the variable will be stored in a register. It means that it MIGHT be stored in a register depending on hardware and

93
93
93
93
93
93
93
93
93
110
93
93
120
93
93

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

930
93
93
93
93
140
93

implementation restrictions.

The static Storage Class

The **static** storage class instructs the compiler to keep a local variable in existence during the life-time of the program instead of creating and destroying it each time it comes into and goes out of scope. Therefore, making local variables static allows them to maintain their values between function calls.

The static modifier may also be applied to global variables. When this is done, it causes that variable's scope to be restricted to the file in which it is declared.

}

94
94
94
94
94
94
94
94
94
110
94
94
120
94
94

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

940
94
94
94
94
140
94

```
printf("i is %d and count is %d\n", i, count);
/* local static variable */
main()
{
   while(count--)
   {
                                func();
   }
   return 0;
```

95
95
95
95
95
95
95
95
95
110
95
95
120
95
95

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

950
95
95
95
95
140
95

```
}
/* function definition */ void func( void )
{
   static int i = 5; i++;
/* global variable */
static int count = 5;
/* function declaration */
void func(void);
#include <stdio.h>
```

96
96
96
96
96
96
96
96
96
110
96
96
120
96
96

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

960
96
96
96
96
140
96

In C programming, when **static** is used on a class data member, it causes only one copy of that member to be shared by all the objects of its class.When the above code is compiled and executed, it produces the following result:

```
i is 6 and count is 4 i
is 7 and count is 3
```

97
97
97
97
97
97
97
97
97
110
97
97
120
97
97

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

970
97
97
97
97
140
97

```
i is 8 and count is 2 i
is 9 and count is 1 i is
10 and count is 0
```

The extern Storage Class

The **extern** storage class is used to give a reference of a global variable that is visible to ALL the program files. When you use 'extern', the variable cannot be initialized, however, it points the variable name at a storage location that has been previously defined.

98
98
98
98
98
98
98
98
98
110
98
98
120
98
98

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

980
98
98
98
98
140
98

When you have multiple files and you define a global variable or function, which will also be used in other files, then *extern* will be used in another file to provide the reference of defined variable or function. Just for understanding, *extern* is used to declare a global variable or function in another file.

The extern modifier is most commonly used when there are two or more files sharing the same global variables or functions as explained below.

```
 #include <stdio.h>


 int count;
```

```
99
99
99
99
99
99
99
99
99
110
99
99
120
99
99
#include <stdio.h>
990
99
99
99
99
140
99
```

```
extern void write_extern();

main()
{
   count = 5;
   write_extern();
}
```

First File: main.c

```
100
100
100
100
100
100
100
100
100
110
100
100
120
100
100
#include <stdio.h>
1000
100
100
100
100
140
100
```

```
#include <stdio.h>

extern int count;

void write_extern(void)
{
```

Second File: support.c

```
101
101
101
101
101
101
101
101
110
101
101
120
101
101
#include <stdio.h>
1010
101
101
101
101
140
101
```

```
   printf("count is %d\n", count);
}
```

```
$gcc main.c support.c
```

Here, *extern* is being used to declare *count* in the second file, whereas it has its definition in the first file, main.c. Now, compile these two files as follows:It will produce the executable program **a.out**. When this program is executed, it produces the following result:

```
5
```

102
102
102
102
102
102
102
102
102
110
102
102
120
102
102

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

1020
102
102
102
102
140
102

Operators

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

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

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

103
103
103
103
103
103
103
103
103
110
103
103
120
103
103
#include <stdio.h>
1030
103
103
103
103
140
103

Arithmetic Operators

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

Operator	Description	Example
+	Adds two operands.	A + B = 30

104
104
104
104
104
104
104
104
104
110
104
104
120
104
104

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

1040
104
104
104
104
140
104

-	Subtracts second operand from the first.	A - B = -10
*	Multiplies both operands.	A * B = 200
/	Divides numerator by de-numerator.	B / A = 2
%	Modulus Operator and remainder of after an integer division.	B % A = 0
++	Increment operator increases the integer value by one.	A++ = 11

105
105
105
105
105
105
105
105
105
110
105
105
120
105
105

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

1050
105
105
105
105
140
105

106
106
106
106
106
106
106
106
106
110
106
106
120
106
106
#include <stdio.h>
1060
106
106
106
106
140
106

--	Decrement operator value by one.	decreases	the	integer	A--	=	9

Example

```
#include <stdio.h>


main()

{
107
107
107
107
107
107
107
107
107
110
107
107
120
107
107
#include <stdio.h>
1070
107
107
107
107
140
107
```

```
    int  a  =  21;
    int  b  =  10;
    int c ;

    c = a + b;
    printf("Line 1 - Value of c is %d\n", c ); c
    = a - b;
    printf("Line 2 - Value of c is %d\n", c ); c
```

108
108
108
108
108
108
108
108
108
110
108
108
120
108
108

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

1080
108
108
108
108
140
108

```
    = a * b;
    printf("Line 3 - Value of c is %d\n", c ); c
    = a / b;
    printf("Line 4 - Value of c is %d\n", c ); c
    = a % b;
    printf("Line 5 - Value of c is %d\n", c ); c
    = a++;
    printf("Line 6 - Value of c is %d\n", c ); c
```

109
109
109
109
109
109
109
109
109
110
109
109
120
109
109

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

1090
109
109
109
109
140
109

```
    = a--;
    printf("Line 7 - Value of c is %d\n", c );


}
```

Try the following example to understand all the arithmetic operators available in C:When you compile and execute the above program, it produces the following result:

```
Line 1 - Value of c is 31
```

110
110
110
110
110
110
110
110
110
110
110
120
110
110

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

1100
110
110
110
110
140
110

ine		Value -	f		s	1
ine		Value -	f		s	10
ine		Value -	f		s	
ine		Value -	f		s	
		-				

111
111
111
111
111
111
111
111
111
110
111
111
120
111
111

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

1110
111
111
111
111
140
111

ine		Value	f		s	1
ine		Value -	f		s	2

Relational Operators

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

Oper	Description	Example

112
112
112
112
112
112
112
112
112
110
112
112
120
112
112

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

1120
112
112
112
112
140
112

ator		
==	Checks if the values of two operands are equal or not. If yes, then the condition becomes true.	(A == B) is not true.
!=	Checks if the values of two operands are equal or not. If the values are not equal, then the condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand. If yes, then the condition	(A > B) is not true.

113
113
113
113
113
113
113
113
113
110
113
113
120
113
113

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

1130
113
113
113
113
140
113

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

114
114
114
114
114
114
114
114
114
110
114
114
120
114
114
#include <stdio.h>
1140
114
114
114
114
140
114

	equal to the value of right operand. If yes, then the condition becomes true.	true.

Example

Try the following example to understand all the relational operators available in C:

#include <stdio.h>

115
115
115
115
115
115
115
115
115
110
115
115
120
115
115

#include <stdio.h>

1150
115
115
115
115
140
115

```
main()
{
int a = 21; int b = 10; int c ;

if( a == b )
{
printf("Line 1 - a is equal to b\n" );
}
else
{
printf("Line 1 - a is not equal to b\n" );
```

```
116
116
116
116
116
116
116
116
116
110
116
116
120
116
116
#include <stdio.h>
1160
116
116
116
116
140
116
```

```
}
if ( a < b )
{
printf("Line 2 - a is less than b\n" );
}
else
{
printf("Line 2 - a is not less than b\n" );
}
if ( a > b )
{
```

117
117
117
117
117
117
117
117
117
110
117
117
120
117
117

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

1170
117
117
117
117
140
117

```
printf("Line 3 - a is greater than b\n" );
}
else
{
```

```
118
118
118
118
118
118
118
118
118
110
118
118
120
118
118
#include <stdio.h>
1180
118
118
118
118
140
118
```

```
        printf("Line 3 - a is not greater than b\n" );
    }
    /* Lets change value of a and b */ a
    = 5;
    b = 20;
    if ( a <= b )
    {
```

119
119
119
119
119
119
119
119
119
110
119
119
120
119
119

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

1190
119
119
119
119
140
119

```
      printf("Line 4 - a is either less than or equal to  b\n" );
   }
   if ( b >= a )
   {
      printf("Line 5 - b is either greater than  or equal to b\n" );
   }
}
Line 1 - a is not equal to b
```

120
120
120
120
120
120
120
120
120
110
120
120
120
120
120

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

1200
120
120
120
120
140
120

```
 Line 2 - a is not less than b

 Line 3 - a is greater than b

 Line 4 - a is either less than or equal to  b Line

 5 - b is either greater than       or equal to b
```

When you compile and execute the above program, it produces the following result:

Logical Operators

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

121
121
121
121
121
121
121
121
121
110
121
121
120
121
121

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

1210
121
121
121
121
140
121

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

122
122
122
122
122
122
122
122
122
110
122
122
120
122
122
#include <stdio.h>
1220
122
122
122
122
140
122

123
123
123
123
123
123
123
123
123
110
123
123
120
123
123
#include <stdio.h>
1230
123
123
123
123
140
123

	becomes true.				
!	Called Logical NOT Operator. It is used to	(A	&	)	s
	reverse the logical state of its operand. If a	rue.			
	condition is true, then Logical NOT operator will				
	make it false.				

124
124
124
124
124
124
124
124
124
110
124
124
120
124
124

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

1240
124
124
124
124
140
124

Example

Try the following example to understand all the logical operators available in C:#include <stdio.h>

```
  main()
 {
    int a = 5;
```

125
125
125
125
125
125
125
125
125
110
125
125
120
125
125

#include <stdio.h>

1250
125
125
125
125
140
125

```
    int b = 20;
    int c ;

    if ( a && b )
    {
       printf("Line 1 - Condition is true\n" );
    }
    if ( a || b )
```

126
126
126
126
126
126
126
126
126
110
126
126
120
126
126
#include <stdio.h>
1260
126
126
126
126
140
126

```
    {
       printf("Line 2 - Condition is true\n" );
    }
    /* lets change the value of  a and b */ a
    = 0;
    b = 10;
    if ( a && b )
    {
```

127
127
127
127
127
127
127
127
127
110
127
127
120
127
127

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

1270
127
127
127
127
140
127

```
      printf("Line 3 - Condition is true\n" );
   }
   else
```

```
128
128
128
128
128
128
128
128
128
110
128
128
120
128
128
#include <stdio.h>
1280
128
128
128
128
140
128
```

```
    {
       printf("Line 3 - Condition is not true\n" );
    }
    if ( !(a && b) )
    {
       printf("Line 4 - Condition is true\n" );
    }
```

129
129
129
129
129
129
129
129
129
110
129
129
120
129
129

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

1290
129
129
129
129
140
129

```
}
Line 1 - Condition is true Line
2 - Condition is true Line 3 -
Condition is not true Line 4 -
Condition is true
```

When you compile and execute the above program, it produces the following result:

Bitwise Operators

Bitwise operators work on bits and perform bit-by-bit operation. The truth table for &, |,

130
130
130
130
130
130
130
130
130
110
130
130
120
130
130

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

1300
130
130
130
130
140
130

and ^ is as follows:

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

131
131
131
131
131
131
131
131
131
110
131
131
120
131
131

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

1310
131
131
131
131
140
131

1	0	0	1	1

Assume A = 60 and B = 13; in binary format, they will be as follows: A = 0011 1100
B = 0000 1101

132
132
132
132
132
132
132
132
132
110
132
132
120
132
132
#include <stdio.h>
1320
132
132
132
132
140
132

A&B = 0000 1100
A|B = 0011 1101
A^B = 0011 0001
~A = 1100 0011

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

Operator	Description	Example

133
133
133
133
133
133
133
133
133
110
133
133
120
133
133

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

1330
133
133
133
133
140
133

&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) = 12, i.e., 0000 1100
|	Binary OR Operator copies a bit if it exists in either operand.	(A | B) = 61, i.e., 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) = 49, i.e., 0011 0001
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) = -61, i.e., 1100 0011 in 2's

134
134
134
134
134
134
134
134
134
110
134
134
120
134
134

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

1340
134
134
134
134
140
134

		complement form.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 = 240, i.e., 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 = 15, i.e., 0000 1111

135
135
135
135
135
135
135
135
135
110
135
135
120
135
135

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

1350
135
135
135
135
140
135

Example

Try the following example to understand all the bitwise operators available in C:

```
#include <stdio.h>

main()
{
```

```
136
136
136
136
136
136
136
136
136
110
136
136
120
136
136
#include <stdio.h>
1360
136
136
136
136
140
136
```

unsigne d int		= 60;	/* 60 = 0011	100	/
unsigne d int int c = 0;		= 13;	/* 13 = 0000	101	/
c = a & b; printf("Line		/ * 12 -	= 0000 1100 */ of c is	);	

137
137
137
137
137
137
137
137
137
110
137
137
120
137
137

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

1370
137
137
137
137
140
137

		Value	%d\n",		
c = a	b; printf("Line		/ * 61 - Value	= 0011 1101 */ of c is %d\n",	);
c = a ^	b; printf("Line		/ * 49 - Value	= 0011 0001 */ of c is %d\n",	);

138
138
138
138
138
138
138
138
138
110
138
138
120
138
138

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

1380
138
138
138
138
140
138

c = ~a; printf("Line		/*-61 - Value	= 1100 0011 */ of c is %d\n",	);	
c = a << 2; printf("Line		/* 240 - Value	= 1111 0000 */ of c is %d\n",	);	

139
139
139
139
139
139
139
139
139
110
139
139
120
139
139

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

1390
139
139
139
139
140
139

```
c = a >> 2;
/* 15 = 0000 1111 */
printf("Line 6 - Value of c is %d\n", c );
}
```

When you compile and execute the above program, it produces the following result:

```
140
140
140
140
140
140
140
140
140
110
140
140
120
140
140
#include <stdio.h>
1400
140
140
140
140
140
140
140
```

ine		- Value	f		s	2
ine		- Value	f		s	1
ine		- Value	f		s	9
ine		- Value	f		s	61
ine		- Value	f		s	40
		-				

141
141
141
141
141
141
141
141
141
110
141
141
120
141
141

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

1410
141
141
141
141
140
141

ine		Value	f		s	5

Assignment Operators

The following tables lists the assignment operators supported by the C language:

Operator	Description	Example
=	Simple assignment operator. Assigns	C = A + B will

```
142
142
142
142
142
142
142
142
142
110
142
142
120
142
142
#include <stdio.h>
1420
142
142
142
142
140
142
```

	values from right side operands to left side operand.	assign the value of A + B to C
+=	Add AND assignment operator. It adds the right operand to the left operand and assigns the result to the left operand.	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator. It subtracts the right operand from the left operand and assigns the result to the left operand.	C -= A is equivalent to C = C - A

143
143
143
143
143
143
143
143
143
110
143
143
120
143
143

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

1430
143
143
143
143
140
143

*=	Multiply AND assignment operator. It multiplies the right operand with the left operand and assigns the result to the left operand.	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator. It divides the left operand with the right operand and assigns the result to the left operand.	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator. It takes modulus using two operands and assigns the result to the left operand.	C %= A is equivalent to C = C % A

144
144
144
144
144
144
144
144
144
110
144
144
120
144
144

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

1440
144
144
144
144
140
144

<<=	Left shift AND assignment operator.	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator.	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator.	C &= 2 is same as C

```
145
145
145
145
145
145
145
145
145
110
145
145
120
145
145
#include <stdio.h>
1450
145
145
145
145
140
145
```

						= C & 2			
^=	Bitwise exclusive operator.	R	nd	assign ment		= C ^		is same as	
|=	Bitwise inclusive operator.	R	nd	assign ment	C	= 2	s	same as C	

146
146
146
146
146
146
146
146
146
110
146
146
120
146
146

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

1460
146
146
146
146
140
146

Example

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

{
   int a = 21; int c ;

c =  a;
```

Try the following example to understand all the assignment operators available in C:

```
147
147
147
147
147
147
147
147
147
110
147
147
120
147
147
#include <stdio.h>
1470
147
147
147
147
140
147
```

148
148
148
148
148
148
148
148
148
110
148
148
120
148
148
#include <stdio.h>
1480
148
148
148
148
140
148

printf("Line		=	Operator	Example,	alue	f			%d\n",		;
c += a; printf("Line		+=	Operator	Example,	alue	f			%d\n",		;
c -= a; printf("Line			Op	Ex					%		

149
149
149
149
149
149
149
149
149
110
149
149
120
149
149

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

1490
149
149
149
149
140
149

		-=	erator	ample,	alue	f			d\n",		;
c *= a; printf("Line		*=	Op erator	Ex ample,	alue	f			% d\n",		;

150
150
150
150
150
150
150
150
150
110
150
150
120
150
150
#include <stdio.h>
1500
150
150
150
150
140
150

c /= a; printf(" Line		- /= Operator Example, Value of c = %d\n", c);
c = 200; c %= a; printf("Line		- %= Operator Example, Value of c = %d\n", c);

151
151
151
151
151
151
151
151
151
110
151
151
120
151
151

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

1510
151
151
151
151
140
151

c <<= 2; printf("Line		- <<= Operator Example, Value of c = %d\n", c);
c >>= 2; printf("Line		- >>= Operator Example, Value of c = %d\n", c);

152
152
152
152
152
152
152
152
152
110
152
152
120
152
152

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

1520
152
152
152
152
140
152

c &= 2; printf("Line		- &= Operator Example, Value of c = %d\n", c);
c ^= 2;		

153
153
153
153
153
153
153
153
153
110
153
153
120
153
153
#include <stdio.h>
1530
153
153
153
153
140
153

```
printf("Line 10 - ^= Operator Example, Value of c = %d\n", c );
c |=  2;
printf("Line 11 - |= Operator Example, Value of c = %d\n", c );
}
```

154
154
154
154
154
154
154
154
154
110
154
154
120
154
154

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

1540
154
154
154
154
140
154

```
Line 7 - <<= Operator Example, Value of c = 44

Line 8 - >>= Operator Example, Value of c = 11
```

When you compile and execute the above program, it produces the following result:

ine		=	erator	Op	ample,	Ex	alue	f	1

```
155
155
155
155
155
155
155
155
155
110
155
155
120
155
155
#include <stdio.h>
1550
155
155
155
155
140
155
```

ine		+=	Op erator	Ex ample,	alue	f			2
ine		-=	Op erator	Ex ample,	alue	f			1
ine		*=	Op erator	Ex ample,	alue	f			41
ine		/=	Op erator	Ex ample,	alue	f			1
ine		%=	Op erator	Ex ample,	alue	f			1

156
156
156
156
156
156
156
156
156
110
156
156
120
156
156

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

1560
156
156
156
156
140
156

```
Line 9 - &= Operator Example, Value of c = 2
Line 10 - ^= Operator Example, Value of c = 0
Line 11 - |= Operator Example, Value of c = 2
```

Misc Operators ↦ sizeof & ternary

Besides the operators discussed above, there are a few other important operators including **sizeof** and **? :** supported by the C Language.

157
157
157
157
157
157
157
157
157
110
157
157
120
157
157
#include <stdio.h>
1570
157
157
157
157
140
157

Operator	Description	Example
sizeof()	Returns the size of a variable.	sizeof(a), where a is integer, will return 4.
&	Returns the address of a variable.	&a; returns the actual address of the variable.

158
158
158
158
158
158
158
158
158
110
158
158
120
158
158

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

1580
158
158
158
158
140
158

*	Pointer to a variable.	*a;
? :	Conditional Expression.	If Condition is true ? then value X : otherwise value Y

Example

```
#include <stdio.h>


main()
```

```
159
159
159
159
159
159
159
159
159
110
159
159
120
159
159
#include <stdio.h>
1590
159
159
159
159
140
159
```

```
{
   int a = 4;
   short b;
   double c;
   int* ptr;
```

Try following example to understand all the miscellaneous operators available in C:

```
160
160
160
160
160
160
160
160
160
110
160
160
120
160
160
#include <stdio.h>
1600
160
160
160
160
140
160
```

```
}
b = (a == 10) ? 20: 30;

printf( "Value of b is %d\n", b );
/* example of ternary operator */

a = 10;

b = (a == 1) ? 20: 30;

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

161
161
161
161
161
161
161
161
161
110
161
161
120
161
161

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

1610
161
161
161
161
140
161

```
printf("*ptr is %d.\n", *ptr);
/* 'ptr' now contains the address of 'a'*/
ptr = &a;
/* example of & and * operators */
/* example of sizeof operator */
printf("Line 1 - Size of variable a = %d\n", sizeof(a) ); printf("Line 2 -
Size of variable b = %d\n", sizeof(b) ); printf("Line 3 - Size of variable c=
%d\n", sizeof(c) );
```

When you compile and execute the above program, it produces the following result:

```
162
162
162
162
162
162
162
162
162
110
162
162
120
162
162
#include <stdio.h>
1620
162
162
162
162
140
162
```

value of a	s	
*ptr is 4.		
Value of b	s	0
Value of b	s	0

Operators Precedence in C

163
163
163
163
163
163
163
163
163
110
163
163
120
163
163

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

1630
163
163
163
163
140
163

Operator precedence determines the grouping of terms in an expression and decides how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has a higher precedence than the addition operator.

For example, x = 7 + 3 * 2; here, x is assigned 13, not 20 because operator * has a higher precedence than +, so it first gets multiplied with 3*2 and then adds into 7.

164
164
164
164
164
164
164
164
164
110
164
164
120
164
164

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

1640
164
164
164
164
140
164

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

Category	Operator	Associativity
Postfix	() [] -> . ++ - -	Left to right
Unary	+ - ! ~ ++ - - (type)* & sizeof	Right to left

165
165
165
165
165
165
165
165
165
110
165
165
120
165
165

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

1650
165
165
165
165
140
165

Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Left to right

166
166
166
166
166
166
166
166
166
110
166
166
120
166
166

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

1660
166
166
166
166
140
166

Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left

167
167
167
167
167
167
167
167
167
110
167
167
120
167
167

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

1670
167
167
167
167
140
167

Assignment	= += -= *= /= %=>>= <<= &= ^= |=	Right to left
Comma	,	Left to right

168
168
168
168
168
168
168
168
168
110
168
168
120
168
168
#include <stdio.h>
1680
168
168
168
168
140
168

Example

Try the following example to understand operator precedence in C:

```
}
return 0;
printf("Value of a + (b * c) / d is  : %d\n" ,  e );
//  20 + (150/5)
e = a + (b * c) / d;
e = (a + b) * (c / d);   // (30) * (15/5)
```

169
169
169
169
169
169
169
169
169
110
169
169
120
169
169

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

1690
169
169
169
169
140
169

```
printf("Value of (a + b) * (c / d) is  : %d\n",  e );
printf("Value of ((a + b) * c) / d is  : %d\n" ,  e );
// (30 * 15 ) / 5
e = ((a + b) * c) / d;
printf("Value of (a + b) * c / d is : %d\n",  e );
// ( 30 * 15 ) / 5
e = (a + b) * c / d;
main()
{
   int a = 20; int b = 10; int c = 15; int d = 5;
```

170
170
170
170
170
170
170
170
170
110
170
170
120
170
170

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

1700
170
170
170
170
140
170

```
   int e;
#include <stdio.h>
```

When you compile and execute the above program, it produces the following result:

```
 Value of (a + b) * c / d is : 90
 Value of ((a + b) * c) / d is: 90
 Value of (a + b) * (c / d) is: 90
 Value of a + (b * c) / d is  : 50
```

171
171
171
171
171
171
171
171
171
110
171
171
120
171
171

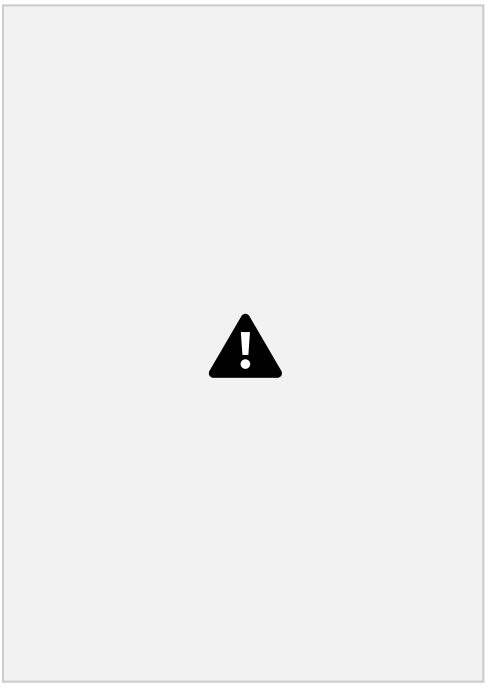
```
#include <stdio.h>
```

1710
171
171
171
171
140
171

Decision

Decision-making structures require that the programmer specifies one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

Shown below is the general form of a typical decision-making structure found in most of the programming languages:



172
172
172
172
172
172
172
172
172
110
172
172
120
172
172

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

1720
172
172
172
172
140
172

C programming language assumes any **non-zero** and **non-null** values as **true**, and if it is either **zero** or **null**, then it is assumed as **false** value.

C programming language provides the following types of decision-making statements.

Statement	Description
if statement	An **if statement** consists of a boolean expression followed by one or more statements.
if...else statement	An **if statement** can be

173
173
173
173
173
173
173
173
173
110
173
173
120
173
173

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

1730
173
173
173
173
140
173

	followed by an optional **else statement**, which executes when

174
174
174
174
174
174
174
174
174
110
174
174
120
174
174

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

1740
174
174
174
174
140
174

	the Boolean expression is false.
nested if statements	You can use one **if** or **else if** statement inside another **if** or **else if** statement(s).
switch statement	A **switch** statement allows a variable to be tested for equality against a list of values.
nested switch statements	You can use one **switch** statement inside another **switch** statement(s).

175
175
175
175
175
175
175
175
175
110
175
175
120
175
175

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

1750
175
175
175
175
140
175

if Statement

An **if** statement consists of a Boolean expression followed by one or more statements.

Syntax

The syntax of an 'if' statement in C programming language is:

```
if(boolean_expression)
{
   /* statement(s) will execute if the boolean expression is true */
```

176
176
176
176
176
176
176
176
176
110
176
176
120
176
176
#include <stdio.h>
1760
176
176
176
176
140
176

```
}
```

If the Boolean expression evaluates to **true**, then the block of code inside the 'if' statement will be executed. If the Boolean expression evaluates to **false**, then the first set of code after the end of the 'if' statement (after the closing curly brace) will be executed.

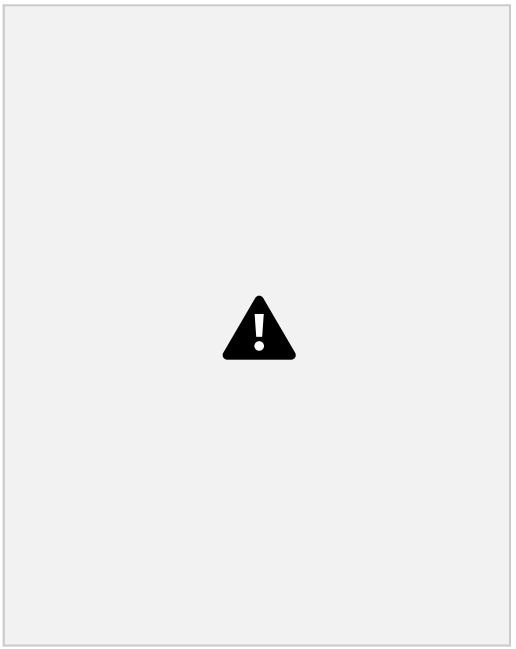
C programming language assumes any **non-zero** and **non-null** values as **true** and if it is either **zero** or **null**, then it is assumed as **false** value.

Flow Diagram

177
177
177
177
177
177
177
177
177
110
177
177
120
177
177

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

1770
177
177
177
177
140
177



178
178
178
178
178
178
178
178
178
110
178
178
120
178
178
#include <stdio.h>
1780
178
178
178
178
140
178

```
#include <stdio.h>

int main ()
{
   /* local variable definition */
   int a = 10;
```

```
179
179
179
179
179
179
179
179
179
110
179
179
120
179
179
#include <stdio.h>
1790
179
179
179
179
140
179
```

```
    /* check the boolean condition using if statement */
    if( a < 20 )
    {
        /* if condition is true then print the following */
        printf("a is less than 20\n" );
    }
    printf("value of a is : %d\n", a);
```

180
180
180
180
180
180
180
180
180
110
180
180
120
180
180
#include <stdio.h>
1800
180
180
180
180
140
180

```
   return 0;
}
```

ExampleWhen the above code is compiled and executed, it produces the following result:

```
a is less than 20;

value of a is : 10
```

if…else Statement

181
181
181
181
181
181
181
181
181
110
181
181
120
181
181

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

1810
181
181
181
181
140
181

An **if** statement can be followed by an optional **else** statement, which executes when the Boolean expression is false.

Syntax

The syntax of an **if...else** statement in C programming language is:

```
if(boolean_expression)
{
    /* statement(s) will execute if the boolean expression is true */
}
```

182
182
182
182
182
182
182
182
182
110
182
182
120
182
182

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

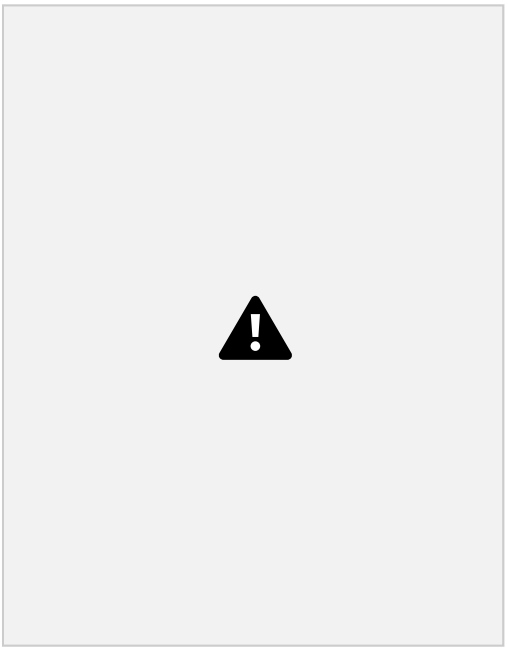
1820
182
182
182
182
140
182

```
else
{
   /* statement(s) will execute if the boolean expression is false */
}
```

If the Boolean expression evaluates to **true**, then the **if block** will be executed, otherwise, the **else block** will be executed.

C programming language assumes any **non-zero** and **non-null** values as **true**, and if it is either **zero** or **null**, then it is assumed as **false** value.

Flow Diagram



183
183
183
183
183
183
183
183
183
110
183
183
120
183
183
#include <stdio.h>
1830
183
183
183
183
140
183

184
184
184
184
184
184
184
184
184
110
184
184
120
184
184
#include <stdio.h>
1840
184
184
184
184
140
184

```
#include <stdio.h>

int main ()
{
   /* local variable definition */
   int a = 100;
```

```
185
185
185
185
185
185
185
185
185
110
185
185
120
185
185
#include <stdio.h>
1850
185
185
185
185
140
185
```

```
    /* check the boolean condition */
    if( a < 20 )
    {
        /* if condition is true then print the following */
        printf("a is less than 20\n" );
    }
    else
    {
```

186
186
186
186
186
186
186
186
186
110
186
186
120
186
186

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

1860
186
186
186
186
140
186

```
        /* if condition is false then print the following */
        printf("a is not less than 20\n" );
    }
    printf("value of a is : %d\n", a);

    return 0;
}
```

ExampleWhen the above code is compiled and executed, it produces the following result:

```
187
187
187
187
187
187
187
187
187
110
187
187
120
187
187
#include <stdio.h>
1870
187
187
187
187
140
187
```

```
a is not less than 20;
value of a is : 100
```

if...else if...else Statement

An **if** statement can be followed by an optional **else if...else** statement, which is very useful to test various conditions using single if...else if statement.

When using if…else if…else statements, there are few points to keep in mind:

- An if can have zero or one else's and it must come after any else if's.
- An if can have zero to many else if's and they must come before the else.

188
188
188
188
188
188
188
188
188
110
188
188
120
188
188

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

1880
188
188
188
188
140
188

189
189
189
189
189
189
189
189
189
110
189
189
120
189
189
#include <stdio.h>
1890
189
189
189
189
140
189

- Once an else if succeeds, none of the remaining else if's or else's will be tested.

Syntax

The syntax of an **if...else if...else** statement in C programming language is:

```
if(boolean_expression 1)
{
   /* Executes when the boolean expression 1 is true */
}
else if( boolean_expression 2)
```

190
190
190
190
190
190
190
190
190
110
190
190
120
190
190
#include <stdio.h>
1900
190
190
190
190
140
190

```
 {
    /* Executes when the boolean expression 2 is true */
 }
 else if( boolean_expression 3)
 {
    /* Executes when the boolean expression 3 is true */
 }
 else
```

191
191
191
191
191
191
191
191
191
110
191
191
120
191
191
```
#include <stdio.h>
```
1910
191
191
191
191
140
191

```
 {
    /* executes when the none of the above condition is true */
 }

 #include <stdio.h>


 int main ()
 {
```

192
192
192
192
192
192
192
192
192
110
192
192
120
192
192

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

1920
192
192
192
192
140
192

```
    /* local variable definition */
    int a = 100;

    /* check the boolean condition */ if(
    a == 10 )
    {
        /* if condition is true then print the following */
```

Example

```
193
193
193
193
193
193
193
193
193
110
193
193
120
193
193
#include <stdio.h>
1930
193
193
193
193
140
193
```

194
194
194
194
194
194
194
194
194
110
194
194
120
194
194
#include <stdio.h>
1940
194
194
194
194
140
194

```
        printf("Value of a is 10\n" );
    }
    else if( a == 20 )
    {
        /* if else if condition is true */
        printf("Value of a is 20\n" );
    }
```

195
195
195
195
195
195
195
195
195
110
195
195
120
195
195
#include <stdio.h>
1950
195
195
195
195
140
195

```
    else if( a == 30 )
    {
        /* if else if condition is true  */
        printf("Value of a is 30\n" );
    }
    else
    {
```

196
196
196
196
196
196
196
196
196
110
196
196
120
196
196
#include <stdio.h>
1960
196
196
196
196
140
196

```
        /* if none of the conditions is true */
        printf("None of the values is matching\n" );
    }
    printf("Exact value of a is: %d\n", a );

    return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
197
197
197
197
197
197
197
197
197
110
197
197
120
197
197
#include <stdio.h>
1970
197
197
197
197
140
197
```

```
None of the values is matching
Exact value of a is: 100
```

Nested if Statements

It is always legal in C programming to **nest** if-else statements, which means you can use one if or else if statement inside another if or else if statement(s).

Syntax

The syntax for a **nested if** statement is as follows:

198
198
198
198
198
198
198
198
198
110
198
198
120
198
198
#include <stdio.h>
1980
198
198
198
198
140
198

```
 if( boolean_expression 1)
 {
```

```
199
199
199
199
199
199
199
199
199
110
199
199
120
199
199
#include <stdio.h>
1990
199
199
199
199
140
199
```

```
    /* Executes when the boolean expression 1 is true */ if(boolean_expression
    2)
    {
       /* Executes when the boolean expression 2 is true */
    }
 }
```

You can nest **else if...else** in the similar way as you have nested *if* statements.

200
200
200
200
200
200
200
200
200
110
200
200
120
200
200

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

2000
200
200
200
200
140
200

```
#include <stdio.h>

int main ()
{
   /* local variable definition */
   int a = 100;
   int b = 200;
```

```
201
201
201
201
201
201
201
201
201
110
201
201
120
201
201
#include <stdio.h>
2010
201
201
201
201
140
201
```

```
    /* check the boolean condition */ if(
    a == 100 )
    {
        /* if condition is true then check the following */
        if( b == 200 )
        {
           /* if condition is true then print the following */
```

202
202
202
202
202
202
202
202
202
110
202
202
120
202
202

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

2020
202
202
202
202
140
202

```
            printf("Value of a is 100 and b is 200\n" );
         }
      }
      printf("Exact value of a is : %d\n", a );
      printf("Exact value of b is : %d\n", b );

      return 0;
   }
```

203
203
203
203
203
203
203
203
203
110
203
203
120
203
203

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

2030
203
203
203
203
140
203

ExampleWhen the above code is compiled and executed, it produces the following result:

```
204
204
204
204
204
204
204
204
204
110
204
204
120
204
204
#include <stdio.h>
2040
204
204
204
204
140
204
```

```
Value of a is 100 and b is 200
Exact value of a is : 100 Exact
value of b is : 200
```

switch Statement

A **switch** statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each **switch case**.

205
205
205
205
205
205
205
205
205
110
205
205
120
205
205

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

2050
205
205
205
205
140
205

Syntax

The syntax for a **switch** statement in C programming language is as follows:

```
}
/* you can have any number of case statements */
default : /* Optional */ statement(s);

:
:
switch(expression){

    case constant-expression statement(s);

       break; /* optional */ case constant-expression
```

206
206
206
206
206
206
206
206
206
110
206
206
120
206
206

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

2060
206
206
206
206
140
206

```
statement(s);

break; /* optional */
```

The following rules apply to a **switch** statement:

· The **expression** used in a **switch** statement must have an integral or enumerated type, or be of a class type in which the class has a single conversion function to an integral or enumerated type.

· You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.

· The **constant-expression** for a case must be the same data type as the variable in the switch, and it must be a constant or a literal.

· When the variable being switched on is equal to a case, the statements following

207
207
207
207
207
207
207
207
110
207
207
120
207
207

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

2070
207
207
207
207
140
207

that case will execute until a **break** statement is reached.

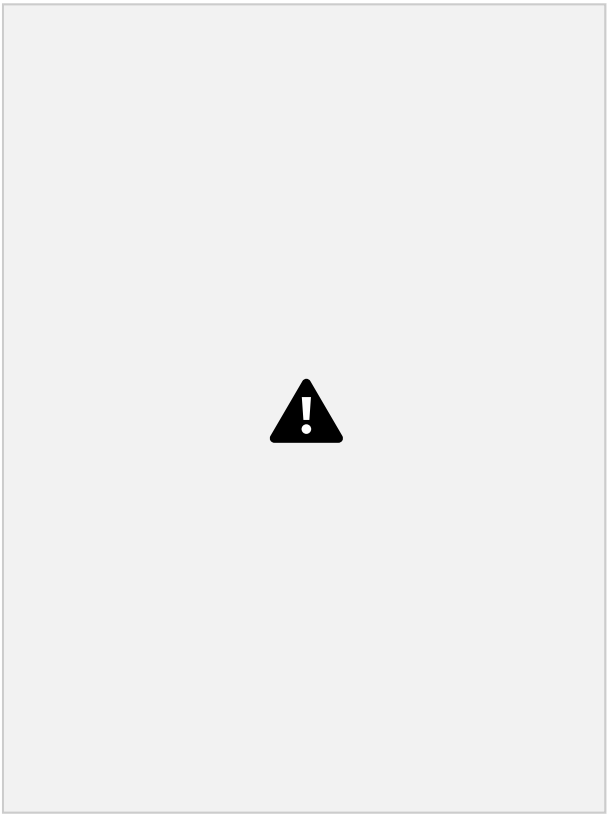
- When a **break** statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.

208
208
208
208
208
208
208
208
208
110
208
208
120
208
208
#include <stdio.h>
2080
208
208
208
208
140
208

· Not every case needs to contain a **break**. If no **break** appears, the flow of control will *fall through* to subsequent cases until a break is reached.

· A **switch** statement can have an optional **default** case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No **break** is needed in the default case.

Flow Diagram



209
209
209
209
209
209
209
209
209
110
209
209
120
209
209

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

2090
209
209
209
209
140
209

```
#include <stdio.h>

int main ()
{
   /* local variable definition */ char
   grade = 'B';
```

210
210
210
210
210
210
210
210
110
210
210
120
210
210

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

2100
210
210
210
210
140
210

```
switch(grade)
{
case 'A' :
```

Example

```
211
211
211
211
211
211
211
211
211
110
211
211
120
211
211
#include <stdio.h>
2110
211
211
211
211
140
211
```

```
      printf("Excellent!\n" );
      break;
   case 'B' :
   case 'C' :
      printf("Well done\n" );
      break;
   case 'D' :
```

212
212
212
212
212
212
212
212
110
212
212
120
212
212

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

2120
212
212
212
212
140
212

```
      printf("You passed\n" );
      break;
   case 'F' :
      printf("Better try again\n" );
      break;
   default :
      printf("Invalid grade\n" );
   }
```

213
213
213
213
213
213
213
213
213
110
213
213
120
213
213

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

2130
213
213
213
213
140
213

```
   printf("Your grade is  %c\n", grade );


   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Well done
Your grade is B
```

Nested switch Statements

214
214
214
214
214
214
214
214
214
110
214
214
120
214
214
#include <stdio.h>
2140
214
214
214
214
140
214

It is possible to have a switch as a part of the statement sequence of an outer switch. Even if the case constants of the inner and outer switch contain common values, no conflicts will arise.

Syntax

The syntax for a **nested switch** statement is as follows:

```
switch(ch1) {
   case 'A':
      printf("This A is part of outer switch" );
```

215
215
215
215
215
215
215
215
215
110
215
215
120
215
215

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

2150
215
215
215
215
140
215

216
216
216
216
216
216
216
216
216
110
216
216
120
216
216
```
#include <stdio.h>
```
2160
216
216
216
216
140
216

```
      switch(ch2) {
         case 'A':
            printf("This A is part of inner switch" );
            break;
         case 'B': /* case code */
      }
      break;
```

217
217
217
217
217
217
217
217
110
217
217
120
217
217

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

2170
217
217
217
217
140
217

```
   case 'B': /* case code */
}

#include <stdio.h>


int main ()
{
   /* local variable definition */
```

218
218
218
218
218
218
218
218
218
110
218
218
120
218
218

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

2180
218
218
218
218
140
218

```
    int a = 100;
    int b = 200;

    switch(a) { case
       100:
          printf("This is part of outer switch\n", a );
          switch(b) {
             case 200:
```

219
219
219
219
219
219
219
219
219
110
219
219
120
219
219
#include <stdio.h>
2190
219
219
219
219
140
219

```
                printf("This is part of inner switch\n", a );
            }
    }
    printf("Exact value of a is : %d\n", a );
    printf("Exact value of b is : %d\n", b );

    return 0;
 }
```

220
220
220
220
220
220
220
220
220
110
220
220
120
220
220

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

2200
220
220
220
220
140
220

Example

When the above code is compiled and executed, it produces the following result:

```
 This is part of outer switch
 This is part of inner switch
 Exact value of a is : 100
 Exact value of b is : 200
```

```
221
221
221
221
221
221
221
221
221
110
221
221
120
221
221
#include <stdio.h>
2210
221
221
221
221
140
221
```

The ? : Operator:

```
Exp1 ? Exp2 : Exp3;
```

We have covered **conditional operator ? :** in the previous chapter which can be used to replace **if...else** statements. It has the following general form:Where Exp1, Exp2, and Exp3 are expressions. Notice the use and placement of the colon.

The value of a ? expression is determined like this:

1. Exp1 is evaluated. If it is true, then Exp2 is evaluated and becomes the value of the entire ? expression.
2. If Exp1 is false, then Exp3 is evaluated and its value becomes the value of the

```
222
222
222
222
222
222
222
222
222
110
222
222
120
222
222
#include <stdio.h>
2220
222
222
222
222
140
222
```

expression.

223
223
223
223
223
223
223
223
223
110
223
223
120
223
223

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

2230
223
223
223
223
140
223

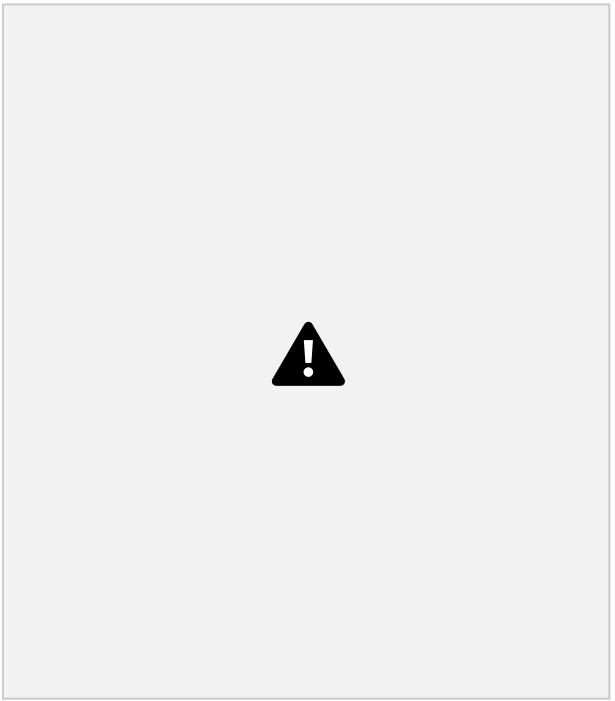
Loops

You may encounter situations when a block of code needs to be executed several number of times. In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times. Given below is the general form of a loop statement in most of the programming languages:

C programming language provides the following types of loops to handle looping



224
224
224
224
224
224
224
224
224
110
224
224
120
224
224
#include <stdio.h>
2240
224
224
224
224
140
224

requirements.

Loop Type	Description
while loop	Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.
for loop	Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.

225
225
225
225
225
225
225
225
225
110
225
225
120
225
225

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

2250
225
225
225
225
140
225

226
226
226
226
226
226
226
226
226
110
226
226
120
226
226
#include <stdio.h>
2260
226
226
226
226
140
226

do...while loop	It is more like a while statement, except that it tests the condition at the end of the loop body.
nested loops	You can use one or more loops inside any other while, for, or do..while loop.

while Loop

A **while** loop in C programming repeatedly executes a target statement as long as a given

227
227
227
227
227
227
227
227
227
110
227
227
120
227
227

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

2270
227
227
227
227
140
227

condition is true.

Syntax

The syntax of a **while** loop in C programming language is:

```
while(condition)
{
   statement(s);
}
```

Here, **statement(s)** may be a single statement or a block of statements. The **condition** may be any expression, and true is any nonzero value. The loop iterates while the condition is

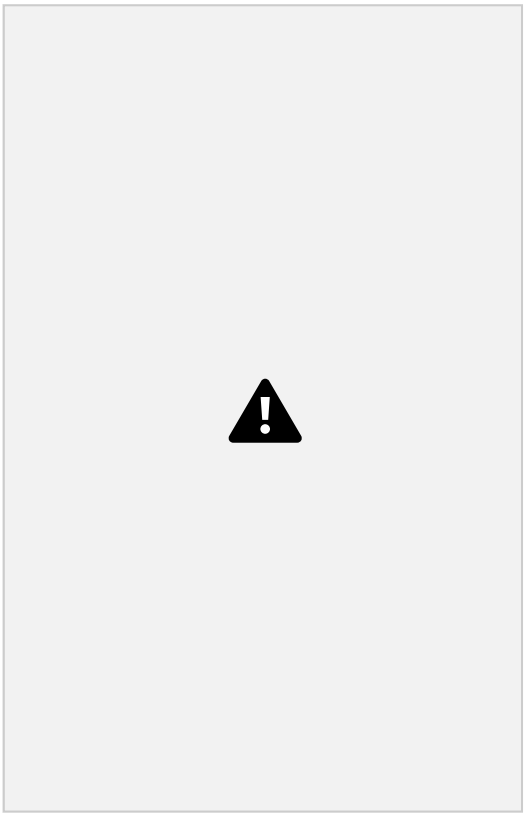
228
228
228
228
228
228
228
228
228
110
228
228
120
228
228
#include <stdio.h>
2280
228
228
228
228
140
228

true.

When the condition becomes false, the program control passes to the line immediately following the loop.

Flow Diagram


```
229
229
229
229
229
229
229
229
229
110
229
229
120
229
229
#include <stdio.h>
2290
229
229
229
229
140
229
```



230
230
230
230
230
230
230
230
230
110
230
230
120
230
230
#include <stdio.h>
2300
230
230
230
230
140
230

Here, the key point to note is that a while loop might not execute at all. When the condition is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

```
 #include <stdio.h>


 int main ()
 {
    /* local variable definition */
```

```
231
231
231
231
231
231
231
231
231
110
231
231
120
231
231
#include <stdio.h>
2310
231
231
231
231
140
231
```

```
    int a = 10;

    /* while loop execution */
    while( a < 20 )
    {
       printf("value of a: %d\n", a);
       a++;
    }
```

232
232
232
232
232
232
232
232
232
110
232
232
120
232
232
#include <stdio.h>
2320
232
232
232
232
140
232

Example

```
233
233
233
233
233
233
233
233
233
110
233
233
120
233
233
#include <stdio.h>
2330
233
233
233
233
140
233
```

```
   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
```

234
234
234
234
234
234
234
234
234
110
234
234
120
234
234

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

2340
234
234
234
234
140
234

```
 value  of  a:  14
 value  of  a:  15
 value  of  a:  16
 value  of  a:  17
 value  of  a:  18
 value of a: 19
```

for Loop

```
235
235
235
235
235
235
235
235
235
110
235
235
120
235
235
#include <stdio.h>
2350
235
235
235
235
140
235
```

A **for** loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

Syntax

The syntax of a **for** loop in C programming language is:

```
for ( init; condition; increment )
{
   statement(s);
}
```

Here is the flow of control in a 'for' loop:

236
236
236
236
236
236
236
236
236
110
236
236
120
236
236

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

2360
236
236
236
236
140
236

1. The **init** step is executed first, and only once. This step allows you to declare and initialize any loop control variables. You are not required to put a statement here, as long as a semicolon appears.

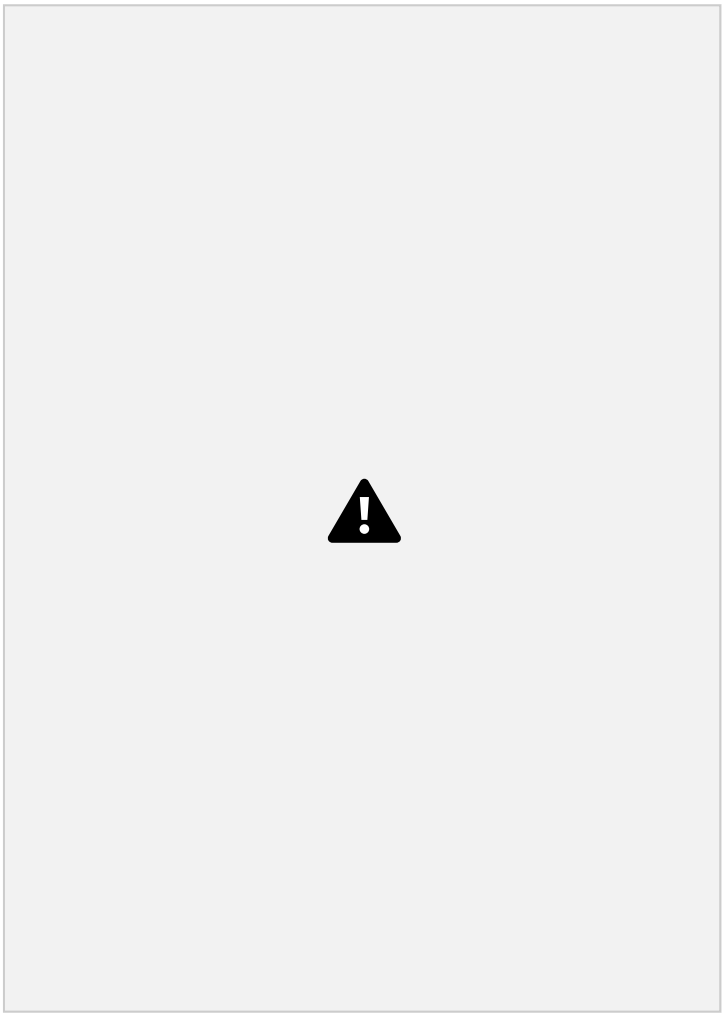
2. Next, the **condition** is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and the flow of control jumps to the next statement just after the 'for' loop.

3. After the body of the 'for' loop executes, the flow of control jumps back up to the **increment** statement. This statement allows you to update any loop control variables. This statement can be left blank, as long as a semicolon appears after the condition.

237
237
237
237
237
237
237
237
237
110
237
237
120
237
237
#include <stdio.h>
2370
237
237
237
237
140
237

4. The condition is now evaluated again. If it is true, the loop executes and the process repeats itself (body of loop, then increment step, and then again condition). After the condition becomes false, the ‘for’ loop terminates.

Flow Diagram



238
238
238
238
238
238
238
238
238
110
238
238
120
238
238

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

2380
238
238
238
238
140
238

```
#include <stdio.h>

int main ()
{
   /* for loop execution */
   for( int a = 10; a < 20; a = a + 1 )
   {
```

```
239
239
239
239
239
239
239
239
239
110
239
239
120
239
239
#include <stdio.h>
2390
239
239
239
239
140
239
```

```
  printf("value of a: %d\n", a);
```

Example

240
240
240
240
240
240
240
240
240
110
240
240
120
240
240

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

2400
240
240
240
240
140
240

```
   }

   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
value of a: 10
value of a: 11
```

241
241
241
241
241
241
241
241
241
110
241
241
120
241
241

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

2410
241
241
241
241
140
241

```
 value  of  a:  12
 value  of  a:  13
 value  of  a:  14
 value  of  a:  15
 value  of  a:  16
 value  of  a:  17
 value  of  a:  18
 value of a: 19
```

242
242
242
242
242
242
242
242
242
110
242
242
120
242
242

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

2420
242
242
242
242
140
242

do…while Loop

Unlike **for** and **while** loops, which test the loop condition at the top of the loop, the **do...while** loop in C programming checks its condition at the bottom of the loop.

A **do...while** loop is similar to a while loop, except the fact that it is guaranteed to execute at least one time.

Syntax

The syntax of a **do...while** loop in C programming language is:

```
do
```

243
243
243
243
243
243
243
243
110
243
243
120
243
243

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

2430
243
243
243
243
140
243

```
{
     statement(s);

}while( condition );
```

Notice that the conditional expression appears at the end of the loop, so the statement(s) in the loop executes once before the condition is tested.

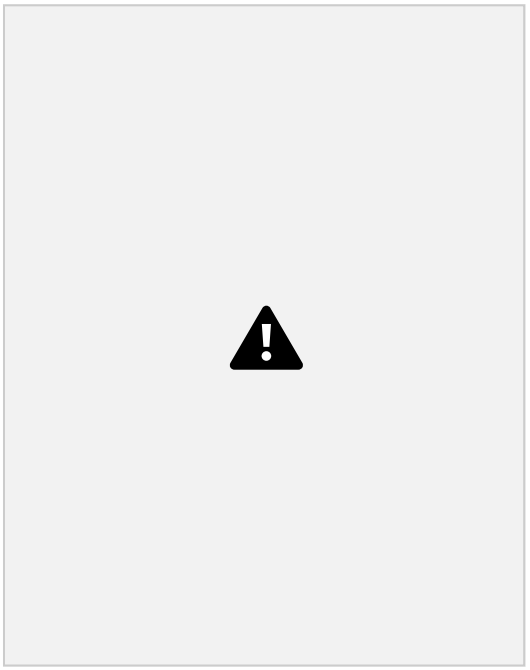
244
244
244
244
244
244
244
244
244
110
244
244
120
244
244

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

2440
244
244
244
244
140
244

If the condition is true, the flow of control jumps back up to do, and the statement(s) in the loop executes again. This process repeats until the given condition becomes false.

Flow Diagram



245
245
245
245
245
245
245
245
245
110
245
245
120
245
245
#include <stdio.h>
2450
245
245
245
245
140
245

```
#include <stdio.h>

int main ()
{
   /* local variable definition */
   int a = 10;
```

246
246
246
246
246
246
246
246
246
110
246
246
120
246
246

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

2460
246
246
246
246
140
246

```
    /* do loop execution */ do
    {
        printf("value of a: %d\n", a); a
        = a + 1;
    }while( a < 20 );

    return 0;
```

Example

```
247
247
247
247
247
247
247
247
247
110
247
247
120
247
247
#include <stdio.h>
2470
247
247
247
247
140
247
```

248
248
248
248
248
248
248
248
248
110
248
248
120
248
248
#include <stdio.h>
2480
248
248
248
248
140
248

```
}
```

When the above code is compiled and executed, it produces the following result:

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
```

249
249
249
249
249
249
249
249
249
110
249
249
120
249
249

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

2490
249
249
249
249
140
249

```
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

Nested Loops

C programming allows to use one loop inside another loop. The following section shows a

```
250
250
250
250
250
250
250
250
250
110
250
250
120
250
250
#include <stdio.h>
2500
250
250
250
250
140
250
```

few examples to illustrate the concept.

Syntax

The syntax for a **nested for loop** statement in C is as follows:

```
for ( init; condition; increment )
{
   for ( init; condition; increment )
   {
      statement(s);
```

251
251
251
251
251
251
251
251
251
110
251
251
120
251
251

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

2510
251
251
251
251
140
251

```
   }
   statement(s);
}
```

The syntax for a **nested while loop** statement in C programming language is as follows:

```
while(condition)
{
   while(condition)
   {
```

252
252
252
252
252
252
252
252
252
110
252
252
120
252
252

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

2520
252
252
252
252
140
252

253
253
253
253
253
253
253
253
253
110
253
253
120
253
253
#include <stdio.h>
2530
253
253
253
253
140
253

```
        statement(s);
    }
    statement(s);
 }
 do
 {
    statement(s); do
```

254
254
254
254
254
254
254
254
254
110
254
254
120
254
254

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

2540
254
254
254
254
140
254

```
    {
        statement(s);
    }while( condition );
```

```
 }while( condition );
```

The syntax for a **nested do...while loop** statement in C programming language is as follows:A final note on loop nesting is that you can put any type of loop inside any other type of loop. For example, a 'for' loop can be inside a 'while' loop or vice versa.

Example

```
255
255
255
255
255
255
255
255
255
110
255
255
120
255
255
#include <stdio.h>
2550
255
255
255
255
140
255
```

```
}
if(j > (i/j)) printf("%d is prime\n", i);
// if factor found, not prime
for(i=2; i<100; i++) {
   for(j=2; j <= (i/j); j++) if(!(i%j)) break;
```

```
int main ()
```

```
{
   /* local variable definition */ int i, j;
```

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

The following program uses a nested for loop to find the prime numbers from 2 to 100:

256
256
256
256
256
256
256
256
256
110
256
256
120
256
256

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

2560
256
256
256
256
140
256

257
257
257
257
257
257
257
257
257
110
257
257
120
257
257
#include <stdio.h>
2570
257
257
257
257
140
257

```
   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

	s	rime
	s	rime

258
258
258
258
258
258
258
258
258
110
258
258
120
258
258

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

2580
258
258
258
258
140
258

	s	rime
	s	rime
1	s	rime
3	s	rime
7	s	rime
9	s	rime

259
259
259
259
259
259
259
259
259
110
259
259
120
259
259

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

2590
259
259
259
259
140
259

3	s	rime
9	s	rime
1	s	rime
7	s	rime
1	s	rime

260
260
260
260
260
260
260
260
260
110
260
260
120
260
260

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

2600
260
260
260
260
140
260

3	s	rime
7	s	rime
3	s	rime
9	s	rime
1	s	rime
7	s	rime

261
261
261
261
261
261
261
261
261
110
261
261
120
261
261

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

2610
261
261
261
261
140
261

1	s	rime
3	s	rime
9	s	rime
3	s	rime
9	s	rime

262
262
262
262
262
262
262
262
262
110
262
262
120
262
262

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

2620
262
262
262
262
140
262

7	s	rime

Loop Control Statements

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

C supports the following control statements.

Control Statement	Description

263
263
263
263
263
263
263
263
263
110
263
263
120
263
263

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

2630
263
263
263
263
140
263

break statement	Terminates the **loop** or **switch** statement and transfers execution to the statement immediately following the loop or switch.
continue statement	Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.
goto statement	Transfers control to the labeled statement.

break Statement

264
264
264
264
264
264
264
264
264
110
264
264
120
264
264

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

2640
264
264
264
264
140
264

The **break** statement in C programming has the following two usages:

· When a **break** statement is encountered inside a loop, the loop is immediately terminated and the program control resumes at the next statement following the loop.

· It can be used to terminate a case in the **switch** statement (covered in the next chapter).

If you are using nested loops, the break statement will stop the execution of the innermost loop and start executing the next line of code after the block.

Syntax

The syntax for a **break** statement in C is as follows:

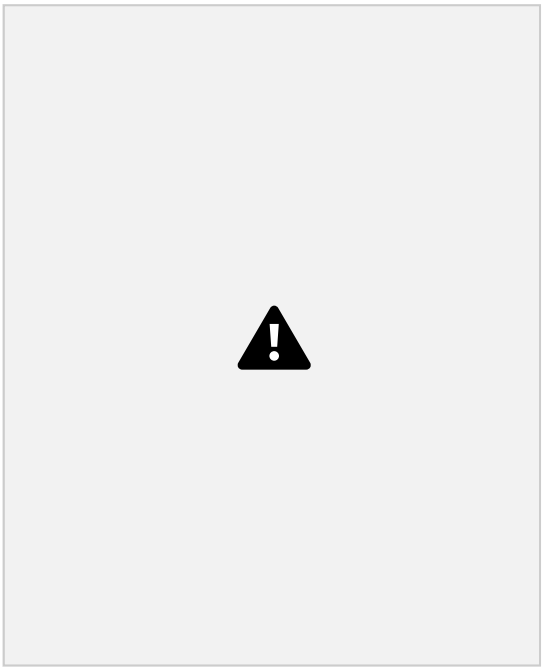
265
265
265
265
265
265
265
265
265
110
265
265
120
265
265
#include <stdio.h>
2650
265
265
265
265
140
265

```
break;
```

Flow Diagram

```
266
266
266
266
266
266
266
266
266
110
266
266
120
266
266
#include <stdio.h>
2660
266
266
266
266
140
266
```

267
267
267
267
267
267
267
267
267
110
267
267
120
267
267
```
#include <stdio.h>
```
2670
267
267
267
267
140
267



268
268
268
268
268
268
268
268
268
268
110
268
268
120
268
268
#include <stdio.h>
2680
268
268
268
268
140
268

```
#include <stdio.h>

int main ()
{
   /* local variable definition */
   int a = 10;
```

269
269
269
269
269
269
269
269
269
110
269
269
120
269
269
#include <stdio.h>
2690
269
269
269
269
140
269

```
    /* while loop execution */
    while( a < 20 )
    {
       printf("value of a: %d\n", a);
       a++;
       if( a > 15)
       {
```

270
270
270
270
270
270
270
270
270
110
270
270
120
270
270
#include <stdio.h>
2700
270
270
270
270
140
270

```
            /* terminate the loop using break statement */
             break;
         }
    }
```

Example

```
271
271
271
271
271
271
271
271
271
110
271
271
120
271
271
#include <stdio.h>
2710
271
271
271
271
140
271
```

```
   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
value of a: 10
value of a: 11
value of a: 12
```

272
272
272
272
272
272
272
272
272
110
272
272
120
272
272

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

2720
272
272
272
272
140
272

```
value of a: 13
value of a: 14
value of a: 15
```

continue Statement

The **continue** statement in C programming works somewhat like the **break** statement. Instead of forcing termination, it forces the next iteration of the loop to take place, skipping any code in between.

For the **for** loop, **continue** statement causes the conditional test and increment portions of

273
273
273
273
273
273
273
273
273
110
273
273
120
273
273

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

2730
273
273
273
273
140
273

the loop to execute. For the **while** and **do...while** loops, **continue** statement causes the program control to pass to the conditional tests.

Syntax

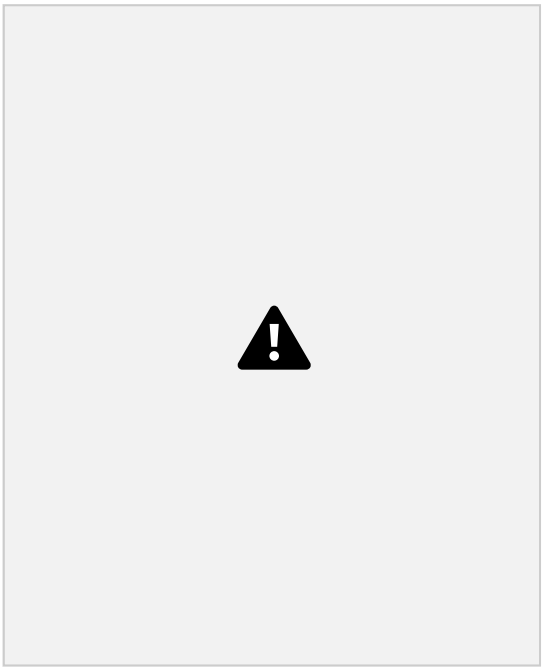
The syntax for a **continue** statement in C is as follows:

```
continue;
```

Flow Diagram


```
274
274
274
274
274
274
274
274
274
110
274
274
120
274
274
#include <stdio.h>
2740
274
274
274
274
140
274
```

275
275
275
275
275
275
275
275
275
110
275
275
120
275
275
#include <stdio.h>
2750
275
275
275
275
140
275



276
276
276
276
276
276
276
276
276
110
276
276
120
276
276
#include <stdio.h>
2760
276
276
276
276
140
276

```
#include <stdio.h>

int main ()
{
   /* local variable definition */
   int a = 10;
```

277
277
277
277
277
277
277
277
277
110
277
277
120
277
277
#include <stdio.h>
2770
277
277
277
277
140
277

```
    /* do loop execution */ do
    {
       if( a == 15)
       {
          /* skip the iteration */ a
          = a + 1;
          continue;
       }
```

278
278
278
278
278
278
278
278
278
110
278
278
120
278
278

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

2780
278
278
278
278
140
278

```
      printf("value of a: %d\n", a);
      a++;

   }while( a < 20 );
```

Example

```
279
279
279
279
279
279
279
279
279
110
279
279
120
279
279
#include <stdio.h>
2790
279
279
279
279
140
279
```

```
   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
value of a: 10
value of a: 11
value of a: 12
```

280
280
280
280
280
280
280
280
280
110
280
280
120
280
280

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

2800
280
280
280
280
140
280

```
value of a: 13
value of a: 14
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

goto Statement

```
281
281
281
281
281
281
281
281
281
110
281
281
120
281
281
#include <stdio.h>
2810
281
281
281
281
140
281
```

A **goto** statement in C programming provides an unconditional jump from the 'goto' to a labeled statement in the same function.

NOTE: Use of **goto** statement is highly discouraged in any programming language because it makes difficult to trace the control flow of a program, making the program hard to understand and hard to modify. Any program that uses a goto can be rewritten to avoid them.

Syntax

The syntax for a **goto** statement in C is as follows:

```
goto label;
..
```

282
282
282
282
282
282
282
282
282
110
282
282
120
282
282

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

2820
282
282
282
282
140
282

.

```
label: statement;
```

Here **label** can be any plain text except C keyword and it can be set anywhere in the C program above or below to **goto** statement.

Flow Diagram

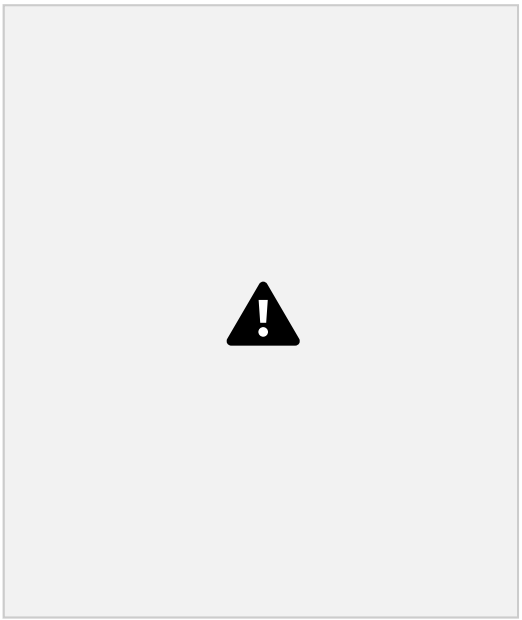
283
283
283
283
283
283
283
283
110
283
283
120
283
283

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

2830
283
283
283
283
140
283

284
284
284
284
284
284
284
284
284
110
284
284
120
284
284
#include <stdio.h>
2840
284
284
284
284
140
284

285
285
285
285
285
285
285
285
285
110
285
285
120
285
285
#include <stdio.h>
2850
285
285
285
285
140
285



286
286
286
286
286
286
286
286
286
110
286
286
120
286
286
#include <stdio.h>
2860
286
286
286
286
140
286

```
#include <stdio.h>

int main ()
{
   /* local variable definition */
   int a = 10;
```

287
287
287
287
287
287
287
287
110
287
287
120
287
287

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

2870
287
287
287
287
140
287

```
    /* do loop execution */
    LOOP:do
    {
       if( a == 15)
       {
          /* skip the iteration */ a
          = a + 1;
          goto LOOP;
```

288
288
288
288
288
288
288
288
288
110
288
288
120
288
288

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

2880
288
288
288
288
140
288

```
        }
        printf("value of a: %d\n", a);
        a++;

    }while( a < 20 );
```

Example

289
289
289
289
289
289
289
289
289
110
289
289
120
289
289

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

2890
289
289
289
289
140
289

```
   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
```

290
290
290
290
290
290
290
290
290
110
290
290
120
290
290

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

2900
290
290
290
290
140
290

```
value of a: 14
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

The Infinite Loop

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

291
291
291
291
291
291
291
291
291
110
291
291
120
291
291

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

2910
291
291
291
291
140
291

```
int main ()
{

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

292
292
292
292
292
292
292
292
292
110
292
292
120
292
292

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

2920
292
292
292
292
140
292

```
    return 0;
}
```

A loop becomes an infinite loop if a condition never becomes false. The **for** loop is traditionally used for this purpose. Since none of the three expressions that form the 'for' loop are required, you can make an endless loop by leaving the conditional expression empty.When the conditional expression is absent, it is assumed to be true. You may have an initialization and increment expression, but C programmers more commonly use the for(;;) construct to signify an infinite loop.

NOTE: You can terminate an infinite loop by pressing Ctrl + C keys.

293
293
293
293
293
293
293
293
293
110
293
293
120
293
293

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

2930
293
293
293
293
140
293

294
294
294
294
294
294
294
294
294
110
294
294
120
294
294
#include <stdio.h>
2940
294
294
294
294
140
294

Functions

A function is a group of statements that together perform a task. Every C program has at least one function, which is **main()**, and all the most trivial programs can define additional functions.

You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division is such that each function performs a specific task.

A function **declaration** tells the compiler about a function's name, return type, and parameters. A function **definition** provides the actual body of the function.

295
295
295
295
295
295
295
295
295
110
295
295
120
295
295

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

2950
295
295
295
295
140
295

The C standard library provides numerous built-in functions that your program can call. For example, **strcat()** to concatenate two strings, **memcpy()** to copy one memory location to another location, and many more functions.

A function can also be referred as a method or a sub-routine or a procedure, etc.

Defining a Function

```
return_type function_name( parameter list )
{
   body of the function
```

```
296
296
296
296
296
296
296
296
296
110
296
296
120
296
296
#include <stdio.h>
2960
296
296
296
296
140
296
```

```
}
```

The general form of a function definition in C programming language is as follows:A function definition in C programming consists of a *function header* and a

function body. Here are all the parts of a function:

· **Return Type**: A function may return a value. The **return_type** is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword **void**.

· **Function Name:** This is the actual name of the function. The function name and the parameter list together constitute the function signature.

· **Parameters:** A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The

297
297
297
297
297
297
297
297
110
297
297
120
297
297

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

2970
297
297
297
297
140
297

parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.

298
298
298
298
298
298
298
298
298
110
298
298
120
298
298
#include <stdio.h>
2980
298
298
298
298
140
298

· **Function Body:** The function body contains a collection of statements that define what the function does.

Example

```
/* function returning the max between two numbers */
int max(int num1, int num2)
{
   /* local variable declaration */ int
   result;
```

299
299
299
299
299
299
299
299
299
110
299
299
120
299
299

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

2990
299
299
299
299
140
299

```
    if (num1 > num2)
       result = num1;
    else
       result = num2;

    return result;
 }
```

Given below is the source code for a function called **max()**. This function takes two

300
300
300
300
300
300
300
300
300
110
300
300
120
300
300

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

3000
300
300
300
300
140
300

parameters num1 and num2 and returns the maximum value between the two:

Function Declarations

A function **declaration** tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately.

A function declaration has the following parts:

```
return_type function_name( parameter list );
```

For the above defined function max(),the function declaration is as follows:

```
int max(int num1, int num2);
```

Parameter names are not important in function declaration, only their type is required, so the following is also a valid declaration:

301
301
301
301
301
301
301
301
301
110
301
301
120
301
301

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

3010
301
301
301
301
140
301

```
int max(int, int);
```

Function declaration is required when you define a function in one source file and you call that function in another file. In such case, you should declare the function at the top of the file calling the function.

302
302
302
302
302
302
302
302
302
110
302
302
120
302
302

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

3020
302
302
302
302
140
302

Calling a Function

While creating a C function, you give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task.

When a program calls a function, the program control is transferred to the called function. A called function performs a defined task and when its return statement is executed or when its function-ending closing brace is reached, it returns the program control back to the main program.

```
#include <stdio.h>


/* function declaration */ int
```

303
303
303
303
303
303
303
303
303
110
303
303
120
303
303

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

3030
303
303
303
303
140
303

```
 max(int num1, int num2);

 int main ()
 {
    /* local variable definition */
    int a = 100;
    int b = 200;
    int ret;
```

304
304
304
304
304
304
304
304
304
110
304
304
120
304
304

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

3040
304
304
304
304
140
304

```
    /* calling a function to get max value */
    ret = max(a, b);

    printf( "Max value is : %d\n", ret );


    return 0;
 }
```

305
305
305
305
305
305
305
305
305
110
305
305
120
305
305

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

3050
305
305
305
305
140
305

```
/* function returning the max between two numbers */ int
max(int num1, int num2)
{
   /* local variable declaration */ int
   result;
```

To call a function, you simply need to pass the required parameters along with the function name, and if the function returns a value, then you can store the returned value. For example:

306
306
306
306
306
306
306
306
306
110
306
306
120
306
306

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

3060
306
306
306
306
140
306

307
307
307
307
307
307
307
307
307
110
307
307
120
307
307
#include <stdio.h>
3070
307
307
307
307
140
307

```
    if (num1 > num2)
       result = num1;
    else
       result = num2;

    return result;
```

308
308
308
308
308
308
308
308
308
110
308
308
120
308
308

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

3080
308
308
308
308
140
308

```
}
Max value is : 200
```

We have kept max()along with main() and compiled the source code. While running the final executable, it would produce the following result:

Function Arguments

If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the **formal parameters** of the function.

Formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.

While calling a function, there are two ways in which arguments can be passed to a

309
309
309
309
309
309
309
309
309
110
309
309
120
309
309

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

3090
309
309
309
309
140
309

function:

Call Type	Description
Call by value	This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.
Call by reference	This method copies the address of an argument into the

310
310
310
310
310
310
310
310
310
110
310
310
120
310
310

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

3100
310
310
310
310
140
310

	formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

311
311
311
311
311
311
311
311
311
110
311
311
120
311
311
#include <stdio.h>
3110
311
311
311
311
140
311

Call by Value

The **call by value** method of passing arguments to a function copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.

```
}
return;
y = temp; /* put temp into y */
/* put y into x */
x = y;
temp = x; /* save the value of x */
/* function definition to swap the values */
```

312
312
312
312
312
312
312
312
312
110
312
312
120
312
312

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

3120
312
312
312
312
140
312

```
void swap(int x, int y)

{

   int temp;
```

By default, C programming uses *call by value* to pass arguments. In general, it means the code within a function cannot alter the arguments used to call the function. Consider the function **swap()** definition as follows.Now, let us call the function **swap()** by passing actual values as in the following example:

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

313
313
313
313
313
313
313
313
313
110
313
313
120
313
313

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

3130
313
313
313
313
140
313

```
 /* function declaration */
 void swap(int x, int y);

 int main ()
 {
    /* local variable definition */
    int a = 100;
    int b = 200;
```

314
314
314
314
314
314
314
314
314
110
314
314
120
314
314

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

3140
314
314
314
314
140
314

```
    printf("Before swap, value of a : %d\n", a );
    printf("Before swap, value of b : %d\n", b );
```

315
315
315
315
315
315
315
315
315
110
315
315
120
315
315

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

3150
315
315
315
315
140
315

```
    /* calling a function to swap the values */
    swap(a, b);

    printf("After swap, value of a : %d\n", a );
    printf("After swap, value of b : %d\n", b );

    return 0;
 }
```

316
316
316
316
316
316
316
316
316
110
316
316
120
316
316

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

3160
316
316
316
316
140
316

```
Before swap, value of a :100
Before swap, value of b :200
After swap, value of a :100
After swap, value of b :200
```

Let us put the above code in a single C file, compile and execute it, it will produce the following result:It shows that there are no changes in the values, though they had been changed inside the function.

Call by Reference

317
317
317
317
317
317
317
317
317
110
317
317
120
317
317

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

3170
317
317
317
317
140
317

The **call by reference** method of passing arguments to a function copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. It means the changes made to the parameter affect the passed argument.

```
/* save the value at address x */

/* put y into x */

/* put temp into y */
temp = *x;

*x = *y;

*y = temp;
```

318
318
318
318
318
318
318
318
318
110
318
318
120
318
318

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

3180
318
318
318
318
140
318

```
/* function definition to swap the values */

void swap(int *x, int *y)

{

   int temp;
```

To pass a value by reference, argument pointers are passed to the functions just like any other value. So accordingly, you need to declare the function parameters as pointer types as in the following function **swap()**, which exchanges the values of the two integer variables pointed to, by their arguments.

319
319
319
319
319
319
319
319
319
110
319
319
120
319
319

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

3190
319
319
319
319
140
319

```
   return;
}
#include <stdio.h>

/* function declaration */
void swap(int *x, int *y);

int main ()
```

320
320
320
320
320
320
320
320
320
110
320
320
120
320
320

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

3200
320
320
320
320
140
320

```
 {
    /* local variable definition */
    int a = 100;
    int b = 200;

    printf("Before swap, value of a : %d\n", a );
    printf("Before swap, value of b : %d\n", b );
```

321
321
321
321
321
321
321
321
321
110
321
321
120
321
321

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

3210
321
321
321
321
140
321

```
    /* calling a function to swap the values.
     * &a indicates pointer to a i.e. address of variable a and
     * &b indicates pointer to b i.e. address of variable b.
    */
    swap(&a, &b);


    printf("After swap, value of a : %d\n", a );
    printf("After swap, value of b : %d\n", b );
```

322
322
322
322
322
322
322
322
322
110
322
322
120
322
322

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

3220
322
322
322
322
140
322

```
    return 0;
}
```

Let us now call the function **swap()** by passing values by reference as in the following example:Let us put the above code in a single C file, compile and execute it, to produce the following result:

```
Before swap, value of a :100
Before swap, value of b :200
```

323
323
323
323
323
323
323
323
323
110
323
323
120
323
323

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

3230
323
323
323
323
140
323

```
After swap, value of a :200

After swap, value of b :100
```

It shows that the change has reflected outside the function as well, unlike call by value where the changes do not reflect outside the function.

By default, C uses **call by value** to pass arguments. In general, it means the code within a function cannot alter the arguments used to call the function.

324
324
324
324
324
324
324
324
324
110
324
324
120
324
324

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

3240
324
324
324
324
140
324

Scope Rules

A scope in any programming is a region of the program where a defined variable can have its existence and beyond that variable it cannot be accessed. There are three places where variables can be declared in C programming language:

- Inside a function or a block which is called **local** variables,
- Outside of all functions which is called **global** variables.
- In the definition of function parameters which are called **formal** parameters.

Let us understand what are **local** and **global** variables, and **formal** parameters.

325
325
325
325
325
325
325
325
325
110
325
325
120
325
325
#include <stdio.h>
3250
325
325
325
325
140
325

```
#include <stdio.h>

int main ()
{
  /* local variable declaration */
  int a, b;
```

326
326
326
326
326
326
326
326
326
110
326
326
120
326
326
#include <stdio.h>
3260
326
326
326
326
140
326

```
   int c;


   /* actual initialization */ a

   = 10;

   b = 20;

   c = a + b;


   printf ("value of a = %d, b = %d and c = %d\n", a, b, c);
```

327
327
327
327
327
327
327
327
327
110
327
327
120
327
327

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

3270
327
327
327
327
140
327

```
   return 0;
```

Variables that are declar

ed inside a function or block are called local variables. They can be used only by statements that are inside that function or block of code. Local variables are not known to functions outside their own. The following example shows how local variables are used. Here all the variables a, b, and c are local to main() function.

328
328
328
328
328
328
328
328
328
110
328
328
120
328
328

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

3280
328
328
328
328
140
328

```
}
```

Global Variables

Global variables are defined outside a function, usually on top of the program. Global variables hold their values throughout the lifetime of your program and they can be accessed inside any of the functions defined for the program.

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

329
329
329
329
329
329
329
329
329
110
329
329
120
329
329

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

3290
329
329
329
329
140
329

```
 /* global variable declaration */ int
 g;

 int main ()
 {
   /* local variable declaration */
   int a, b;
```

330
330
330
330
330
330
330
330
330
110
330
330
120
330
330

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

3300
330
330
330
330
140
330

```
   /* actual initialization */ a
   = 10;
   b = 20;
   g = a + b;

   printf ("value of a = %d, b = %d and g = %d\n", a, b, g);

   return 0;
```

331
331
331
331
331
331
331
331
331
110
331
331
120
331
331

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

3310
331
331
331
331
140
331

```
}
```

A global variable can be accessed by any function. That is, a global variable is available for use throughout your entire program after its declaration. The following program shows how global variables are used in a program.A program can have same name for local and global variables but the value of local variable inside a function will take preference. Here is an example:

```
#include <stdio.h>


/* global variable declaration */ int

g = 20;
```

332
332
332
332
332
332
332
332
332
110
332
332
120
332
332

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

3320
332
332
332
332
140
332

333
333
333
333
333
333
333
333
333
110
333
333
120
333
333
#include <stdio.h>
3330
333
333
333
333
140
333

```
 int main ()
 {
   /* local variable declaration */
   int g = 10;

   printf ("value of g = %d\n",  g);
```

334
334
334
334
334
334
334
334
334
110
334
334
120
334
334

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

3340
334
334
334
334
140
334

```
   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
value of g = 10
```

Formal Parameters

```
return 0;
c);
a);
```

335
335
335
335
335
335
335
335
335
110
335
335
120
335
335

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

3350
335
335
335
335
140
335

```
printf ("value of a in main() = %d\n",

c = sum( a, b);

printf ("value of c in main() = %d\n",
int main ()

{

  /* local variable declaration in main function */ int a = 10;

  int b = 20;

  int c = 0;
/* global variable declaration */
```

336
336
336
336
336
336
336
336
336
110
336
336
120
336
336

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

3360
336
336
336
336
140
336

```
int a = 20;
#include <stdio.h>
```

Formal parameters are treated as local variables with-in a function and they take precedence over global variables. Following is an example:

```
337
337
337
337
337
337
337
337
337
110
337
337
120
337
337
#include <stdio.h>
3370
337
337
337
337
140
337
```

```
}
/* function to add two integers */
int sum(int a, int b)
{
    printf ("value of a in sum() = %d\n",  a);
    printf ("value of b in sum() = %d\n",  b);
    return a + b;
}
```

When the above code is compiled and executed, it produces the following result:

```
338
338
338
338
338
338
338
338
338
110
338
338
120
338
338
#include <stdio.h>
3380
338
338
338
338
140
338
```

```
value of a in main() = 10
value of a in sum() = 10
value of b in sum() = 20
value of c in main() = 30
```

Initializing Local and Global Variables When a local variable is defined, it is not initialized by the system, you must initialize it yourself. Global variables are initialized automatically by the system when you define them, as follows:

Data Type	Initial Default Value

339
339
339
339
339
339
339
339
339
110
339
339
120
339
339

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

3390
339
339
339
339
140
339

int	0
char	'\0'
float	0
double	0
pointer	NULL

340
340
340
340
340
340
340
340
340
110
340
340
120
340
340

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

3400
340
340
340
340
140
340

341
341
341
341
341
341
341
341
341
110
341
341
120
341
341
```
#include <stdio.h>
```
3410
341
341
341
341
140
341

It is a good programming practice to initialize variables properly, otherwise your program may produce unexpected results, because uninitialized variables will take some garbage value already available at their memory location.

342
342
342
342
342
342
342
342
342
110
342
342
120
342
342
```
#include <stdio.h>
```
3420
342
342
342
342
140
342

Arrays

Arrays a kind of data structure that can store a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables. A specific element in an array is accessed by an index.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.

343
343
343
343
343
343
343
343
343
110
343
343
120
343
343

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

3430
343
343
343
343
140
343

Declaring Arrays

```
type arrayName [ arraySize ];
```

To declare an array in C, a programmer specifies the type of the elements and the number of elements required by an array as follows:This is called a *single-dimensional* array. The **arraySize** must be an integer constant greater than zero and **type** can be any valid C data type. For example, to declare a 10-element array called **balance** of type double, use this statement:

```
double balance[10];
```

Here, *balance* is a variable array which is sufficient to hold up to 10 double numbers.

344
344
344
344
344
344
344
344
344
110
344
344
120
344
344

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

3440
344
344
344
344
140
344

Initializing Arrays

```
double balance[5] = {1000.0, 2.0, 3.4, 7.0, 50.0};
```

You can initialize an array in C either one by one or using a single statement as follows:

345
345
345
345
345
345
345
345
345
110
345
345
120
345
345

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

3450
345
345
345
345
140
345

The number of values between braces { } cannot be larger than the number of elements that we declare for the array between square brackets [ ].

```
double balance[] = {1000.0, 2.0, 3.4, 7.0, 50.0};
```

If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write:You will create exactly the same array as you did in the previous example. Following is an example to assign a single element of the array:

```
balance[4] = 50.0;
```

The above statement assigns the 5th element in the array with a value of 50.0. All arrays have 0 as the index of their first element which is also called the base index and the last index of an array will be total size of the array minus 1. Shown below is the pictorial representation of the array we discussed above:

346
346
346
346
346
346
346
346
346
110
346
346
120
346
346

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

3460
346
346
346
346
140
346

Accessing Array Elements

```
 double salary = balance[9];
```

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array. For example:The above statement

347
347
347
347
347
347
347
347
347
110
347
347
120
347
347

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

3470
347
347
347
347
140
347

will take the 10th element from the array and assign the value to salary variable. The following example shows how to use all the three above-mentioned concepts viz. declaration, assignment, and accessing arrays:

```
/* set element at location i to i + 100 */
n[ i ] = i + 100;
/* initialize elements of array n to 0 */

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

{
int main ()

{
```

348
348
348
348
348
348
348
348
348
110
348
348
120
348
348

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

3480
348
348
348
348
140
348

```
   int n[ 10 ]; /* n is an array of 10 integers */ int i,j;
#include <stdio.h>
```

349
349
349
349
349
349
349
349
349
110
349
349
120
349
349

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

3490
349
349
349
349
140
349

```
    }

    /* output each array element's value */
    for (j = 0; j < 10; j++ )
    {
       printf("Element[%d] = %d\n", j, n[j] );
    }
```

350
350
350
350
350
350
350
350
350
110
350
350
120
350
350

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

3500
350
350
350
350
140
350

```
   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

Element[0]		00
Element[1]		01
Elem		

351
351
351
351
351
351
351
351
351
110
351
351
120
351
351

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

3510
351
351
351
351
140
351

ent[2]		02
Element[3]		03
Element[4]		04
Element[5]		05
Element[6]		06
Element[7]		07

352
352
352
352
352
352
352
352
352
110
352
352
120
352
352

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

3520
352
352
352
352
140
352

Element[8]		08
Element[9]		09

Arrays in Detail

Arrays are important to C and should need a lot more attention. The following important concepts related to array should be clear to a C programmer:

353
353
353
353
353
353
353
353
353
110
353
353
120
353
353

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

3530
353
353
353
353
140
353

Concept	Description
Multidimensional arrays	C supports multidimensional arrays. The simplest form of the multidimensional array is the two-dimensional array.
Passing arrays to functions	You can pass to the function a pointer to an array by specifying the array's name without an

	index.

354
354
354
354
354
354
354
354
354
110
354
354
120
354
354

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

3540
354
354
354
354
140
354

Return array from a function	C allows a function to return an array.
Pointer to an array	You can generate a pointer to the first element of an array by simply specifying the array name, without any index.

MultidimensionalArrays

```
type name[size1][size2]...[sizeN];
```

C programming language allows multidimensional arrays. Here is the general form of a

355
355
355
355
355
355
355
355
355
110
355
355
120
355
355

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

3550
355
355
355
355
140
355

multidimensional array declaration:For example, the following declaration creates a three-dimensional integer array:

```
int threedim[5][10][4];
```

Two-dimensionalArrays

```
type arrayName [ x ][ y ];
```

The simplest form of multidimensional array is the two-dimensional array. A two-dimensional array is, in essence, a list of one-dimensional arrays. To declare a two-dimensional integer array of size [x][y], you would write something as follows:Where **type** can be any valid C data type and **arrayName** will be a valid C identifier. A two-dimensional array can be considered as a table which will have x number of rows and y number of columns. A

356
356
356
356
356
356
356
356
356
110
356
356
120
356
356

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

3560
356
356
356
356
140
356

two-dimensional array **a**, which contains three rows and four columns can be shown as follows:

Thus, every element in the array **a** is identified by an element name of the form **a[i][j]**, where ‘a’ is the name of the array, and ‘i' and ‘j’ are the subscripts that uniquely identify each element in ‘a’.

357
357
357
357
357
357
357
357
357
110
357
357
120
357
357
```
#include <stdio.h>
```
3570
357
357
357
357
140
357

358
358
358
358
358
358
358
358
358
110
358
358
120
358
358
```
#include <stdio.h>
```
3580
358
358
358
358
140
358

Initializing Two-Dimensional Arrays

```
initializers for row indexed by 0 */
initializers for row indexed by 1 */ initializers for row indexed by 2 */

/*

/*

/*
                                int a[3][4] = {

                                 {0, 1, 2, 3} ,

                                 {4, 5, 6, 7} ,
```

359
359
359
359
359
359
359
359
359
110
359
359
120
359
359

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

3590
359
359
359
359
140
359

```
                                {8, 9, 10, 11}
};
```

Multidimensional arrays may be initialized by specifying bracketed values for each row. Following is an array with 3 rows and each row has 4 columns.The nested braces, which indicate the intended row, are optional. The following initialization is equivalent to the previous example:

```
int a[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
```

Accessing Two-Dimensional Array Elements

```
int val = a[2][3];
```

An element in a two-dimensional array is accessed by using the subscripts, i.e., row index

360
360
360
360
360
360
360
360
360
110
360
360
120
360
360

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

3600
360
360
360
360
140
360

and column index of the array. For example:The above statement will take the 4th element from the 3rd row of the array. You can verify it in the above figure. Let us check the following program where we have used a nested loop to handle a two-dimensional array:

```
#include <stdio.h>


int main ()
{
   /* an array with 5 rows and 2 columns*/
   int a[5][2] = { {0,0}, {1,2}, {2,4}, {3,6},{4,8}};
```

361
361
361
361
361
361
361
361
361
110
361
361
120
361
361

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

3610
361
361
361
361
140
361

```
    int i, j;

    /* output each array element's value */ for
    ( i = 0; i < 5; i++ )
    {
       for ( j = 0; j < 2; j++ )
       {
          printf("a[%d][%d] = %d\n", i,j, a[i][j] );
```

362
362
362
362
362
362
362
362
362
110
362
362
120
362
362

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

3620
362
362
362
362
140
362

363
363
363
363
363
363
363
363
363
110
363
363
120
363
363
```
#include <stdio.h>
```
3630
363
363
363
363
140
363

```
      }
   }
   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
a[0][0]: 0
a[0][1]: 0
a[1][0]: 1
```

364
364
364
364
364
364
364
364
364
110
364
364
120
364
364

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

3640
364
364
364
364
140
364

```
 a[1][1]: 2
 a[2][0]: 2
 a[2][1]: 4
 a[3][0]: 3
 a[3][1]: 6
 a[4][0]: 4
 a[4][1]: 8
```

As explained above, you can have arrays with any number of dimensions, although it is

365
365
365
365
365
365
365
365
110
365
365
120
365
365

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

3650
365
365
365
365
140
365

likely that most of the arrays you create will be of one or two dimensions.

Passing Arrays to Functions

If you want to pass a single-dimension array as an argument in a function, you would have to declare a formal parameter in one of following three ways and all three declaration methods produce similar results because each tells the compiler that an integer pointer is going to be received. Similarly, you can pass multi-dimensional arrays as formal parameters.

Way-1

Formal parameters as a pointer:

```
void myFunction(int *param)
```

366
366
366
366
366
366
366
366
110
366
366
120
366
366

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

3660
366
366
366
366
140
366

{

.

.

.

}

367
367
367
367
367
367
367
367
110
367
367
120
367
367

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

3670
367
367
367
367
140
367

Way-2

Formal parameters as a sized array:

```
void myFunction(int param[10])
{
.
.
.
}
```

368
368
368
368
368
368
368
368
368
110
368
368
120
368
368

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

3680
368
368
368
368
140
368

Way-3

Formal parameters as an unsized array:

```
void myFunction(int param[])
{
.
.
.
}
```

369
369
369
369
369
369
369
369
369
110
369
369
120
369
369

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

3690
369
369
369
369
140
369

Example

```
avg = sum / size;
for (i = 0; i < size; ++i)
{
  sum += arr[i];
}
double avg;
double sum;
i;
```

370
370
370
370
370
370
370
370
370
110
370
370
120
370
370

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

3700
370
370
370
370
140
370

```
int
double getAverage(int arr[], int size)
{
```

Now, consider the following function, which takes an array as an argument along with another argument and based on the passed arguments, it returns the average of the numbers passed through the array as follows:

371
371
371
371
371
371
371
371
371
110
371
371
120
371
371

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

3710
371
371
371
371
140
371

```
   return avg;
}
```

Now, let us call the above function as follows:

```
#include <stdio.h>

/* function declaration */
```

372
372
372
372
372
372
372
372
372
110
372
372
120
372
372

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

3720
372
372
372
372
140
372

```
double getAverage(int arr[], int size);

int main ()
{
   /* an int array with 5 elements */
   int balance[5] = {1000, 2, 3, 17, 50};
   double avg;
```

373
373
373
373
373
373
373
373
373
110
373
373
120
373
373

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

3730
373
373
373
373
140
373

```
    /* pass pointer to the array as an argument */ avg
    = getAverage( balance, 5 ) ;

    /* output the returned value */
    printf( "Average value is: %f ", avg );

    return 0;
 }
```

When the above code is compiled together and executed, it produces the following result:

```
374
374
374
374
374
374
374
374
374
110
374
374
120
374
374
#include <stdio.h>
3740
374
374
374
374
140
374
```

```
Average value is: 214.400000
```

As you can see, the length of the array doesn't matter as far as the function is concerned because C performs no bounds checking for formal parameters.

Return Array from a Function

C programming does not allow to return an entire array as an argument to a function. However, you can return a pointer to an array by specifying the array's name without an index.

375
375
375
375
375
375
375
375
375
110
375
375
120
375
375

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

3750
375
375
375
375
140
375

```
int * myFunction()
{
.
.
.
}
```

If you want to return a single-dimension array from a function, you would have to declare a function returning a pointer as in the following example:Second point to remember is that C does not advocate to return the address of a local variable to outside of the function, so you would

376
376
376
376
376
376
376
376
376
110
376
376
120
376
376

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

3760
376
376
376
376
140
376

have to define the local variable as **static** variable.

```
#include <stdio.h>


/* function to generate and return random numbers */ int
* getRandom( )
{
  static int  r[10];
  int i;
```

377
377
377
377
377
377
377
377
377
110
377
377
120
377
377

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

3770
377
377
377
377
140
377

```
   /* set the seed */
   srand( (unsigned)time( NULL ) );
   for ( i = 0; i < 10; ++i)
   {
      r[i] = rand();
      printf( "r[%d] = %d\n", i, r[i]);
```

378
378
378
378
378
378
378
378
378
110
378
378
120
378
378

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

3780
378
378
378
378
140
378

```
   }

   return r;
}

/* main function to call above defined function */
```

Now, consider the following function which will generate 10 random numbers and return them using an array and call this function as follows:

```
379
379
379
379
379
379
379
379
379
110
379
379
120
379
379
#include <stdio.h>
3790
379
379
379
379
140
379
```

```
 int main ()
 {
    /* a pointer to an int */ int
    *p;
    int i;


    p = getRandom();
```

380
380
380
380
380
380
380
380
380
110
380
380
120
380
380

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

3800
380
380
380
380
140
380

```
    for ( i = 0; i < 10; i++ )
    {
        printf( "*(p + %d) : %d\n", i, *(p + i));
    }

    return 0;
}
```

r[0] = 313959809

381
381
381
381
381
381
381
381
381
110
381
381
120
381
381

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

3810
381
381
381
381
140
381

r[1] = 1759055877

r[2] = 1113101911

r[3] = 2133832223

r[4] = 2073354073

r[5] = 167288147

r[6] = 1827471542

r[7] = 834791014

r[8] = 1901409888

382
382
382
382
382
382
382
382
382
110
382
382
120
382
382

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

3820
382
382
382
382
140
382

r[9] = 1990469526

*(p + 0) : 313959809

*(p + 1) : 1759055877

*(p + 2) : 1113101911

*(p + 3) : 2133832223

*(p + 4) : 2073354073

*(p + 5) : 167288147

*(p + 6) : 1827471542

383
383
383
383
383
383
383
383
383
110
383
383
120
383
383

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

3830
383
383
383
383
140
383

```
*(p + 7) : 834791014
```

When the above code is compiled together and executed, it produces the following result:

```
384
384
384
384
384
384
384
384
384
110
384
384
120
384
384
#include <stdio.h>
3840
384
384
384
384
140
384
```

```
*(p + 8) : 1901409888
*(p + 9) : 1990469526
```

Pointer to an Array

It is most likely that you would not understand this section until you are through with the chapter 'Pointers'.

```
double balance[50];
```

Assuming you have some understanding of pointers in C, let us start: An array name is a constant pointer to the first element of the array. Therefore, in the declaration:**balance** is a pointer

385
385
385
385
385
385
385
385
385
110
385
385
120
385
385

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

3850
385
385
385
385
140
385

to &balance[0], which is the address of the first element of the array balance. Thus, the following program fragment assigns **p** as the address of the first element of **balance**:

```
double *p;
double balance[10];


p = balance;
```

It is legal to use array names as constant pointers, and vice versa. Therefore, *(balance + 4) is a legitimate way of accessing the data at balance[4].

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

386
386
386
386
386
386
386
386
386
110
386
386
120
386
386

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

3860
386
386
386
386
140
386

```
 int main ()
 {
    /* an array with 5 elements */
    double balance[5] = {1000.0, 2.0, 3.4, 17.0, 50.0};
    double *p;
    int i;
```

387
387
387
387
387
387
387
387
387
110
387
387
120
387
387

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

3870
387
387
387
387
140
387

```
p = balance;


/* output each array element's value */
```

Once you store the address of the first element in ‘p’, you can access the array elements using *p, *(p+1), *(p+2), and so on. Given below is the example to show all the concepts discussed above:

```
388
388
388
388
388
388
388
388
388
110
388
388
120
388
388
#include <stdio.h>
3880
388
388
388
388
140
388
```

```
    printf( "Array values using pointer\n");
    for ( i = 0; i < 5; i++ )
    {
        printf("*(p + %d) : %f\n",  i, *(p + i) );
    }

    printf( "Array values using balance as address\n");
```

389
389
389
389
389
389
389
389
389
110
389
389
120
389
389

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

3890
389
389
389
389
140
389

```
    for ( i = 0; i < 5; i++ )
    {
        printf("*(balance + %d) : %f\n",  i, *(balance + i) );
    }

    return 0;
 }
```

Array values using pointer
Array values using balance as address

390
390
390
390
390
390
390
390
390
110
390
390
120
390
390

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

3900
390
390
390
390
140
390

When the above code is compiled and executed, it produces the following result:

(p	0) :	-1000.000000
(p	1) :	-2.000000
(p	2) :	-3.400000
(p	3) :	-17.000000

391
391
391
391
391
391
391
391
391
110
391
391
120
391
391

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

3910
391
391
391
391
140
391

(p	4) :	-50.000000

*(balance	0) :	-1000.000000
*(balance	1) :	-2.000000
*(balance	2) :	-3.400000
*(b		-17.000

392
392
392
392
392
392
392
392
392
110
392
392
120
392
392

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

3920
392
392
392
392
140
392

alance	3) :	000
*(b alance	- 4) :	50.000 000

In the above example, p is a pointer to double, which means it can store the address of a variable of double type. Once we have the address in p, ***p** will give us the value available at the address stored in p, as we have shown in the above example.

393
393
393
393
393
393
393
393
393
110
393
393
120
393
393

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

3930
393
393
393
393
140
393

Pointers

Pointers in C are easy and fun to learn. Some C programming tasks are performed more easily with pointers, and other tasks, such as dynamic memory allocation, cannot be performed without using pointers. So it becomes necessary to learn pointers to become a perfect C programmer. Let's start learning them in simple and easy steps.

```
#include <stdio.h>


int main ()

{
```

394
394
394
394
394
394
394
394
394
110
394
394
120
394
394

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

3940
394
394
394
394
140
394

```
    int  var1; char
    var2[10];

    printf("Address of var1 variable: %x\n", &var1  );
    printf("Address of var2 variable: %x\n", &var2  );

    return 0;
 }
```

As you know, every variable is a memory location and every memory location has its

395
395
395
395
395
395
395
395
395
110
395
395
120
395
395

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

3950
395
395
395
395
140
395

address defined which can be accessed using ampersand (&) operator, which denotes an address in memory. Consider the following example, which prints the address of the variables defined:When the above code is compiled and executed, it produces the following result:

```
Address of var1 variable: bff5a400
Address of var2 variable: bff5a3f6
```

What are Pointers?

A **pointer** is a variable whose value is the address of another variable, i.e., direct address

396
396
396
396
396
396
396
396
396
110
396
396
120
396
396

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

3960
396
396
396
396
140
396

of the memory location. Like any variable or constant, you must declare a pointer before using it to store any variable address. The general form of a pointer variable declaration is:

```
 type *var-name;
/* pointer to an integer */

/* pointer to a double */

/* pointer to a float */

/* pointer to a character */
double *dp;
```

397
397
397
397
397
397
397
397
397
110
397
397
120
397
397

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

3970
397
397
397
397
140
397

```
float  *fp; char  *ch

*ip;
int
```

Here, **type** is the pointer's base type; it must be a valid C data type and **var- name** is the name of the pointer variable. The asterisk * used to declare a pointer is the same asterisk used for multiplication. However, in this statement, the asterisk is being used to designate a variable as a pointer. Take a look at some of the valid pointer declarations:The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, a long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

398
398
398
398
398
398
398
398
398
110
398
398
120
398
398

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

3980
398
398
398
398
140
398

How to Use Pointers?

```
/* address stored in pointer variable */

printf("Address stored in ip variable: %x\n", ip );
printf("Address of var variable: %x\n", &var  );
/* store address of var in pointer variable*/
ip = &var;
/* actual variable declaration */

/* pointer variable declaration */
int main ()


399
399
399
399
399
399
399
399
399
110
399
399
120
399
399
#include <stdio.h>
3990
399
399
399
399
140
399
```

```
{
   int  var = 20; int   *ip;
```

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

There are a few important operations, which we will do with the help of pointers very frequently. **(a)** We define a pointer variable, **(b)** assign the address of a variable to a pointer, and **(c)** finally access the value at the address available in the pointer variable. This is done by using unary operator * that returns the value of the variable located at the address specified by its operand. The following example makes use of these operations:

400
400
400
400
400
400
400
400
400
110
400
400
120
400
400

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

4000
400
400
400
400
140
400

```
   /* access the value using the pointer */
   printf("Value of *ip variable: %d\n", *ip );

   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Address of var variable: bffd8b3c Address
```

401
401
401
401
401
401
401
401
401
110
401
401
120
401
401

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

4010
401
401
401
401
140
401

```
stored in ip variable: bffd8b3c Value of
*ip variable: 20
```

NULL Pointers

It is always a good practice to assign a NULL value to a pointer variable in case you do not have an exact address to be assigned. This is done at the time of variable declaration. A pointer that is assigned NULL is called a **null** pointer.

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

402
402
402
402
402
402
402
402
402
110
402
402
120
402
402

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

4020
402
402
402
402
140
402

```
int main ()
{
   int  *ptr = NULL;

   printf("The value of ptr is : %x\n", ptr  );

   return 0;
```

403
403
403
403
403
403
403
403
403
110
403
403
120
403
403

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

4030
403
403
403
403
140
403

```
}
```

The NULL pointer is a constant with a value of zero defined in several standard libraries. Consider the following program:When the above code is compiled and executed, it produces the following result:

```
The value of ptr is 0
```

In most of the operating systems, programs are not permitted to access memory at address 0 because that memory is reserved by the operating system. However, the memory address 0 has special significance; it signals that the pointer is not intended to point to an accessible memory location. But by convention, if a pointer contains the null (zero) value, it is assumed to point to nothing.

404
404
404
404
404
404
404
404
404
110
404
404
120
404
404

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

4040
404
404
404
404
140
404

405
405
405
405
405
405
405
405
405
110
405
405
120
405
405
#include <stdio.h>
4050
405
405
405
405
140
405

To check for a null pointer, you can use an ‘if’ statement as follows:

```
/* succeeds if p is not null */

/* succeeds if p is null */
if(ptr)

if(!ptr)
```

Pointers in Detail

Pointers have many but easy concepts and they are very important to C programming. The following important pointer concepts should be clear to any C programmer:

406
406
406
406
406
406
406
406
406
110
406
406
120
406
406
#include <stdio.h>
4060
406
406
406
406
140
406
406

Concept	Description
Pointer arithmetic	There are four arithmetic operators that can be used in pointers: ++, --, +, -
Array of pointers	You can define arrays to hold a number of pointers.
Pointer to pointer	C allows you to have pointer on a pointer and so on.

407
407
407
407
407
407
407
407
407
110
407
407
120
407
407

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

4070
407
407
407
407
140
407

Passing pointers to functions in C	Passing an argument by reference or by address enable the passed argument to be changed in the calling function by the called function.
Return pointer from functions in C	C allows a function to return a pointer to the local variable, static variable, and dynamically allocated memory as well.

Pointer Arithmetic

A pointer in C is an address, which is a numeric value. Therefore, you can perform

408
408
408
408
408
408
408
408
408
110
408
408
120
408
408

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

4080
408
408
408
408
140
408

arithmetic operations on a pointer just as you can on a numeric value. There are four arithmetic operators that can be used on pointers: ++, --, +, and

-

To understand pointer arithmetic, let us consider that **ptr** is an integer pointer which points to the address 1000. Assuming 32-bit integers, let us perform the following arithmetic operation on the pointer:

409
409
409
409
409
409
409
409
409
110
409
409
120
409
409

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

4090
409
409
409
409
140
409

```
ptr++
```

After the above operation, the **ptr** will point to the location 1004 because each time ptr is incremented, it will point to the next integer location which is 4 bytes next to the current location. This operation will move the pointer to the next memory location without impacting the actual value at the memory location. If **ptr** points to a character whose address is 1000, then the above operation will point to the location 1001 because the next character will be available at 1001.

Incrementing a Pointer

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

410
410
410
410
410
410
410
410
410
110
410
410
120
410
410

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

4100
410
410
410
410
140
410

```
const int MAX = 3;

int main ()
{
   int  var[] = {10, 100, 200};
   int  i, *ptr;
```

411
411
411
411
411
411
411
411
411
110
411
411
120
411
411
#include <stdio.h>
4110
411
411
411
411
140
411

```
    /* let us have array address in pointer */ ptr
    = var;
    for ( i = 0; i < MAX; i++)
    {

       printf("Address of var[%d] = %x\n", i, ptr );
       printf("Value of var[%d] = %d\n", i, *ptr );
```

412
412
412
412
412
412
412
412
412
110
412
412
120
412
412

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

4120
412
412
412
412
140
412

```
        /* move to the next location */
        ptr++;
    }
    return 0;
}
```

We prefer using a pointer in our program instead of an array because the variable pointer can be incremented, unlike the array name which cannot be incremented because it is a constant pointer. The following program increments the variable pointer to access each succeeding element of the array:

```
413
413
413
413
413
413
413
413
413
110
413
413
120
413
413
#include <stdio.h>
4130
413
413
413
413
140
413
```

When the above code is compiled and executed, it produces the following result:

```
Address of var[0] = bf882b30
Value of var[0] = 10  Address
of var[1] = bf882b34 Value of
var[1] = 100
Address of var[2] = bf882b38
Value of var[2] = 200
```

414
414
414
414
414
414
414
414
414
110
414
414
120
414
414

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

4140
414
414
414
414
140
414

Decrementing a Pointer

```
#include <stdio.h>

const int MAX = 3;

int main ()
{
   int  var[] = {10, 100, 200};
```

415
415
415
415
415
415
415
415
415
110
415
415
120
415
415
#include <stdio.h>
4150
415
415
415
415
140
415

```
    int  i, *ptr;

    /* let us have array address in pointer */ ptr
    = &var[MAX-1];
    for ( i = MAX; i > 0; i--)
    {
```

416
416
416
416
416
416
416
416
416
110
416
416
120
416
416

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

4160
416
416
416
416
140
416

```
      printf("Address of var[%d] = %x\n", i, ptr );
      printf("Value of var[%d] = %d\n", i, *ptr );

      /* move to the previous location */
      ptr--;
   }
   return 0;
}
```

The same considerations apply to decrementing a pointer, which decreases its value by the

417
417
417
417
417
417
417
417
110
417
417
120
417
417

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

4170
417
417
417
417
140
417

number of bytes of its data type as shown below:

418
418
418
418
418
418
418
418
418
110
418
418
120
418
418
#include <stdio.h>
4180
418
418
418
418
140
418

When the above code is compiled and executed, it produces the following result:

```
Address of var[3] = bfedbcd8
Value of var[3] = 200 Address
of var[2] = bfedbcd4 Value of
var[2] = 100 Address of var[1]
= bfedbcd0 Value of var[1] =
10
```

419
419
419
419
419
419
419
419
419
110
419
419
120
419
419

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

4190
419
419
419
419
140
419

Pointer Comparisons

Pointers may be compared by using relational operators, such as ==, <, and >. If p1 and p2 point to variables that are related to each other, such as elements of the same array, then p1 and p2 can be meaningfully compared.

```
#include <stdio.h>


const int MAX = 3;


int main ()
```

```
420
420
420
420
420
420
420
420
420
110
420
420
120
420
420
#include <stdio.h>
4200
420
420
420
420
140
420
```

```
 {
    int  var[] = {10, 100, 200};
    int  i, *ptr;

    /* let us have address of the first element in pointer */ ptr
    = var;
    i = 0;
    while ( ptr <= &var[MAX - 1] )
```

421
421
421
421
421
421
421
421
421
110
421
421
120
421
421

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

4210
421
421
421
421
140
421

```
    {

        printf("Address of var[%d] = %x\n", i, ptr );
        printf("Value of var[%d] = %d\n", i, *ptr );

        /* point to the previous location */
        ptr++;
```

The following program modifies the previous example - one by incrementing the variable

422
422
422
422
422
422
422
422
422
110
422
422
120
422
422
#include <stdio.h>
4220
422
422
422
422
140
422

pointer so long as the address to which it points is either less than or equal to the address of the last element of the array, which is &var[MAX - 1]:

423
423
423
423
423
423
423
423
423
110
423
423
120
423
423
#include <stdio.h>
4230
423
423
423
423
140
423

```
        i++;
    }
    return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Address of var[0] = bfdbcb20
Value of var[0] = 10  Address
```

424
424
424
424
424
424
424
424
424
110
424
424
120
424
424
#include <stdio.h>
4240
424
424
424
424
140
424

```
of var[1] = bfdbcb24 Value of
var[1] = 100 Address of var[2]
= bfdbcb28 Value of var[2] =
200
```

Array of Pointers

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

425
425
425
425
425
425
425
425
425
110
425
425
120
425
425

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

4250
425
425
425
425
140
425

```
 const int MAX = 3;

 int main ()
 {
    int  var[] = {10, 100, 200};
    int i;

    for (i = 0; i < MAX; i++)
```

426
426
426
426
426
426
426
426
426
110
426
426
120
426
426

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

4260
426
426
426
426
140
426

```
    {
       printf("Value of var[%d] = %d\n", i, var[i] );
    }
    return 0;
 }
```

Before we understand the concept of arrays of pointers, let us consider the following example, which uses an array of 3 integers:When the above code is compiled and executed, it produces the following result:

```
 Value of var[0] = 10
```

427
427
427
427
427
427
427
427
427
110
427
427
120
427
427

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

4270
427
427
427
427
140
427

428
428
428
428
428
428
428
428
428
110
428
428
120
428
428
```
#include <stdio.h>
```
4280
428
428
428
428
140
428

```
Value of var[1] = 100

Value of var[2] = 200

int *ptr[MAX];
```

There may be a situation when we want to maintain an array, which can store pointers to an int or char or any other data type available. Following is the declaration of an array of pointers to an integer:It declares **ptr** as an array of MAX integer pointers. Thus, each element in ptr holds a pointer to an int value. The following example uses three integers, which are stored in an array of pointers, as follows:

429
429
429
429
429
429
429
429
429
110
429
429
120
429
429

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

4290
429
429
429
429
140
429

```
#include <stdio.h>

const int MAX = 3;

int main ()
{
   int  var[] = {10, 100, 200};
   int i, *ptr[MAX];
```

430
430
430
430
430
430
430
430
430
110
430
430
120
430
430

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

4300
430
430
430
430
140
430

```
    for ( i = 0; i < MAX; i++)
    {
       ptr[i] = &var[i]; /* assign the address of integer. */
    }
    for ( i = 0; i < MAX; i++)
    {
       printf("Value of var[%d] = %d\n", i, *ptr[i] );
```

431
431
431
431
431
431
431
431
431
110
431
431
120
431
431

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

4310
431
431
431
431
140
431

```
   }
   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Value of var[0] = 10
Value of var[1] = 100
Value of var[2] = 200
```

432
432
432
432
432
432
432
432
432
110
432
432
120
432
432

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

4320
432
432
432
432
140
432

433
433
433
433
433
433
433
433
433
110
433
433
120
433
433
```
#include <stdio.h>
```
4330
433
433
433
433
140
433

```
#include <stdio.h>

const int MAX = 4;

int main ()
{
   char *names[] = {
```

434
434
434
434
434
434
434
434
110
434
434
120
434
434

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

4340
434
434
434
434
140
434

```
                    "Zara    Ali",
                    "Hina    Ali",
                    "Nuha    Ali",
                    "Sara Ali",
    };
    int i = 0;

    for ( i = 0; i < MAX; i++)
```

435
435
435
435
435
435
435
435
435
110
435
435
120
435
435

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

4350
435
435
435
435
140
435

```
    {
       printf("Value of names[%d] = %s\n", i, names[i] );
    }
    return 0;
}
```

You can also use an array of pointers to character to store a list of strings as follows:When the above code is compiled and executed, it produces the following result:

```
Value of names[0] = Zara Ali
```

436
436
436
436
436
436
436
436
436
110
436
436
120
436
436

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

4360
436
436
436
436
140
436

```
Value of names[1] = Hina Ali
Value of names[2] = Nuha Ali
Value of names[3] = Sara Ali
```

Pointer to Pointer

A pointer to a pointer is a form of multiple indirection, or a chain of pointers. Normally, a pointer contains the address of a variable. When we define a pointer to a pointer, the first pointer contains the address of the second pointer, which points to the location that contains the actual value as shown below.

437
437
437
437
437
437
437
437
437
110
437
437
120
437
437

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

4370
437
437
437
437
140
437

438
438
438
438
438
438
438
438
438
110
438
438
120
438
438
#include <stdio.h>
4380
438
438
438
438
140
438

```
int **var;
```

A variable that is a pointer to a pointer must be declared as such. This is done by placing an additional asterisk in front of its name. For example, the following declaration declares a pointer to a pointer of type int:When a target value is indirectly pointed to by a pointer to a pointer, accessing that value requires that the asterisk operator be applied twice, as is shown below in the example:

```
439
439
439
439
439
439
439
439
439
110
439
439
120
439
439
#include <stdio.h>
4390
439
439
439
439
140
439
```

```
#include <stdio.h>

int main ()
{
   int  var;
   int  *ptr; int
        **pptr;
```

440
440
440
440
440
440
440
440
110
440
440
120
440
440

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

4400
440
440
440
440
140
440

```
    var = 3000;

    /* take the address of var */
    ptr = &var;

    /* take the address of ptr using address of operator & */ pptr
    = &ptr;
```

441
441
441
441
441
441
441
441
441
110
441
441
120
441
441

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

4410
441
441
441
441
140
441

```
    /* take the value using pptr */
    printf("Value of var = %d\n", var );
    printf("Value available at *ptr = %d\n", *ptr );
    printf("Value available at **pptr = %d\n", **pptr);

    return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
442
442
442
442
442
442
442
442
442
110
442
442
120
442
442
#include <stdio.h>
4420
442
442
442
442
140
442
```

443
443
443
443
443
443
443
443
443
110
443
443
120
443
443
#include <stdio.h>
4430
443
443
443
443
140
443

```
Value of var = 3000
Value available at *ptr = 3000 Value
available at **pptr = 3000
```

Passing Pointers to Functions

C programming allows passing a pointer to a function. To do so, simply declare the function parameter as a pointer type.

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

444
444
444
444
444
444
444
444
110
444
444
120
444
444

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

4440
444
444
444
444
140
444

```
#include <time.h>

void getSeconds(unsigned long *par);


int main ()
{
   unsigned long sec;
```

445
445
445
445
445
445
445
445
445
110
445
445
120
445
445

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

4450
445
445
445
445
140
445

```
    getSeconds( &sec );

    /* print the actual value */ printf("Number
    of seconds: %ld\n", sec );

    return 0;
 }
```

446
446
446
446
446
446
446
446
446
110
446
446
120
446
446

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

4460
446
446
446
446
140
446

```
void getSeconds(unsigned long *par)
{
   /* get the current number of seconds */
   *par = time( NULL );
   return;
}
```

Following is a simple example where we pass an unsigned long pointer to a function and change the value inside the function which reflects back in the calling function:

```
447
447
447
447
447
447
447
447
447
110
447
447
120
447
447
#include <stdio.h>
4470
447
447
447
447
140
447
```

448
448
448
448
448
448
448
448
448
110
448
448
120
448
448
#include <stdio.h>
4480
448
448
448
448
140
448

When the above code is compiled and executed, it produces the following result:

```
Number of seconds :1294450468
```

The function, which can accept a pointer, can also accept an array as shown in the following example:

```
#include <stdio.h>


/* function declaration */
double getAverage(int *arr, int size);
```

449
449
449
449
449
449
449
449
449
110
449
449
120
449
449

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

4490
449
449
449
449
140
449

```
int main ()
{
/* an int array with 5 elements */ int balance[5] = {1000, 2, 3, 17, 50}; double avg;

/* pass pointer to the array as an argument */ avg = getAverage( balance, 5 ) ;

/* output the returned value                    */ printf("Average value is: %f\n", avg
);

return 0;
```

450
450
450
450
450
450
450
450
450
110
450
450
120
450
450

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

4500
450
450
450
450
140
450

```
}

double getAverage(int *arr, int size)
{
int                    i, sum = 0; double avg;

for (i = 0; i < size; ++i)
{
sum += arr[i];
}
```

451
451
451
451
451
451
451
451
451
110
451
451
120
451
451

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

4510
451
451
451
451
140
451

452
452
452
452
452
452
452
452
452
110
452
452
120
452
452
```
#include <stdio.h>
```
4520
452
452
452
452
140
452

```
   avg = (double)sum / size;


   return avg;
}
Average value is: 214.40000
```

When the above code is compiled together and executed, it produces the following result:

Return Pointer from Functions

```
int * myFunction()
```

453
453
453
453
453
453
453
453
453
110
453
453
120
453
453

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

4530
453
453
453
453
140
453

```
{
.
.
.
}
```

We have seen in the last chapter how C programming allows to return an array from a function. Similarly, C also allows to return a pointer from a function. To do so, you would have to declare a function returning a pointer as in the following example:Second point to remember is that, it is not a good idea to return the address of a local variable outside the function, so you

```
454
454
454
454
454
454
454
454
454
110
454
454
120
454
454
#include <stdio.h>
4540
454
454
454
454
140
454
```

would have to define the local variable as **static** variable.

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

 /* function to generate and retrun random numbers. */ int
 * getRandom( )
 {
    static int  r[10];
```

455
455
455
455
455
455
455
455
455
110
455
455
120
455
455

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

4550
455
455
455
455
140
455

```
    int i;
```

Now, consider the following function which will generate 10 random numbers and return them using an array name which represents a pointer, i.e., address of first array element.

456
456
456
456
456
456
456
456
456
110
456
456
120
456
456
#include <stdio.h>
4560
456
456
456
456
140
456

```
    /* set the seed */
    srand( (unsigned)time( NULL ) );
    for ( i = 0; i < 10; ++i)
    {
       r[i] = rand();
       printf("%d\n", r[i] );
    }
```

```
457
457
457
457
457
457
457
457
457
110
457
457
120
457
457
#include <stdio.h>
4570
457
457
457
457
140
457
```

```
   return r;
}

/* main function to call above defined function */
int main ()
{
```

458
458
458
458
458
458
458
458
458
110
458
458
120
458
458
#include <stdio.h>
4580
458
458
458
458
140
458

```
    /* a pointer to an int */ int
    *p;
    int i;


    p = getRandom();
    for ( i = 0; i < 10; i++ )
    {
        printf("*(p + [%d]) : %d\n", i, *(p + i) );
```

459
459
459
459
459
459
459
459
459
110
459
459
120
459
459

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

4590
459
459
459
459
140
459

```
    }

    return 0;
 }
 1523198053
 1187214107
 1108300978
 430494959
```

460
460
460
460
460
460
460
460
460
110
460
460
120
460
460

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

4600
460
460
460
460
140
460

```
1421301276

930971084
```

When the above code is compiled together and executed, it produces the following result:

```
461
461
461
461
461
461
461
461
461
110
461
461
120
461
461
#include <stdio.h>
4610
461
461
461
461
140
461
```

123250484

106932140

1604461820

149169022

462
462
462
462
462
462
462
462
462
110
462
462
120
462
462
#include <stdio.h>
4620
462
462
462
462
140
462

(p		0]) :	[1523 198053
(p		1]) :	[1187 214107
(p		2]) :	[1108 300978
(p		3]) :	[4304 94959

463
463
463
463
463
463
463
463
463
110
463
463
120
463
463

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

4630
463
463
463
463
140
463

(p		[4]) :	1421 301276
(p		[5]) :	9309 71084
(p		[6]) :	1232 50484
(p		[7]) :	1069 32140
(p		[8]) :	1604 461820
		[	1491

464
464
464
464
464
464
464
464
464
110
464
464
120
464
464

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

4640
464
464
464
464
140
464

(p		9]) :	69022

465
465
465
465
465
465
465
465
465
110
465
465
120
465
465

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

4650
465
465
465
465
140
465

Strings

Strings are actually one-dimensional array of characters terminated by a **null** character '\0'. Thus a null-terminated string contains the characters that comprise the string followed by a **null**.

```
char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

The following declaration and initialization create a string consisting of the word "Hello". To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word "Hello."If you follow the rule of array initialization, then you can write the above statement as follows:

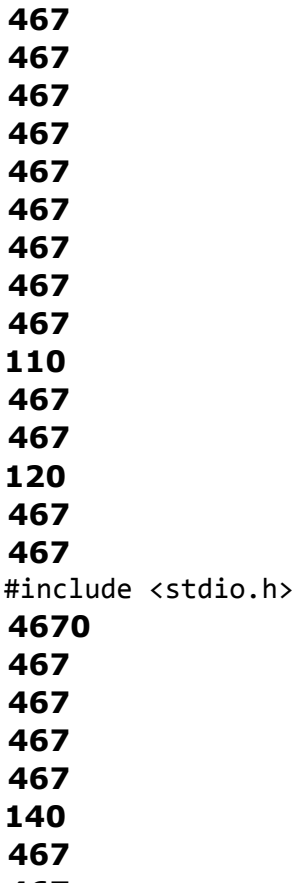
```
char greeting[] = "Hello";
```

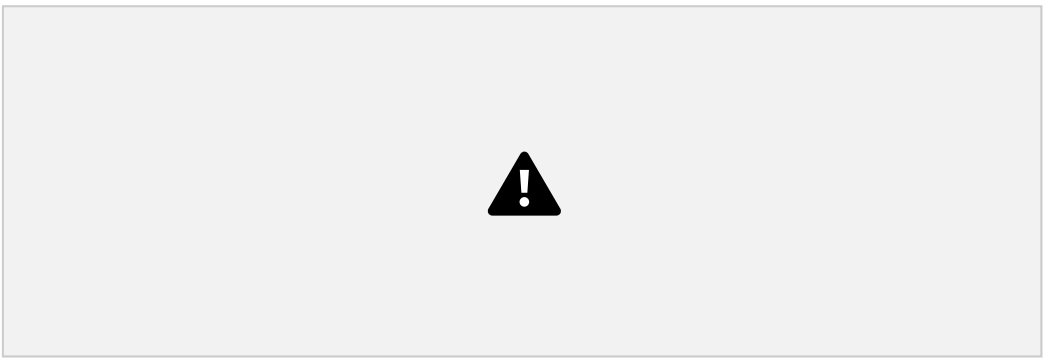
466
466
466
466
466
466
466
466
466
110
466
466
120
466
466

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

4660
466
466
466
466
140
466

Following is the memory presentation of the above defined string in C/C++:





468
468
468
468
468
468
468
468
468
110
468
468
120
468
468

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

4680
468
468
468
468
140
468

Actually, you do not place the *null* character at the end of a string constant. The C compiler automatically places the '\0' at the end of the string when it initializes the array. Let us try to print the above mentioned string:

```
#include <stdio.h>


int main ()
{
   char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

469
469
469
469
469
469
469
469
469
110
469
469
120
469
469

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

4690
469
469
469
469
140
469

```
    printf("Greeting message: %s\n", greeting );
```

470
470
470
470
470
470
470
470
470
110
470
470
120
470
470

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

4700
470
470
470
470
140
470

```
   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Greeting message: Hello
```

C supports a wide range of functions that manipulate null-terminated strings:

471
471
471
471
471
471
471
471
471
110
471
471
120
471
471
#include <stdio.h>
4710
471
471
471
471
140
471

.N.	Function & Purpose
	strcpy(s1, s2); Copies string s2 into string s1.
	strcat(s1, s2); Concatenates string s2 onto the end of string s1.
	strlen(s1); Returns the length of string s1.

472
472
472
472
472
472
472
472
472
110
472
472
120
472
472
#include <stdio.h>
4720
472
472
472
472
140
472

	strcmp(s1, s2); Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2.
	strchr(s1, ch); Returns a pointer to the first occurrence of character ch in string s1.
	strstr(s1, s2); Returns a pointer to the first occurrence of string s2 in string s1.

473
473
473
473
473
473
473
473
473
110
473
473
120
473
473
#include <stdio.h>
4730
473
473
473
473
140
473

The following example uses some of the above-mentioned functions:

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

int main ()
```

```
474
474
474
474
474
474
474
474
474
110
474
474
120
474
474
#include <stdio.h>
4740
474
474
474
474
140
474
```

```
 {
    char  str1[12]  =  "Hello";
    char  str2[12]  =  "World";
    char str3[12];
    int  len ;

    /* copy str1 into str3 */
```

475
475
475
475
475
475
475
475
475
110
475
475
120
475
475

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

4750
475
475
475
475
140
475

```
    strcpy(str3, str1);
    printf("strcpy( str3, str1) :  %s\n", str3 );

    /* concatenates str1 and str2 */
    strcat( str1, str2);
    printf("strcat( str1, str2):   %s\n", str1 );

    /* total lenghth of str1 after concatenation */
```

476
476
476
476
476
476
476
476
476
110
476
476
120
476
476

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

4760
476
476
476
476
140
476

```
   len = strlen(str1);
   printf("strlen(str1) :  %d\n", len );

   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
strcpy( str3, str1) :  Hello
strcat( str1, str2):   HelloWorld
```

477
477
477
477
477
477
477
477
477
110
477
477
120
477
477

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

4770
477
477
477
477
140
477

```
 strlen(str1) :  10
```

```
478
478
478
478
478
478
478
478
478
110
478
478
120
478
478
#include <stdio.h>
4780
478
478
478
478
140
478
```

Structures

Arrays allow to define type of variables that can hold several data items of the same kind. Similarly, **structure** is another user-defined data type available in C that allows to combine data items of different kinds.

Structures are used to represent a record. Suppose you want to keep track of your books in a library. You might want to track the following attributes about each book:

- Title
- Author
- Subject
- Book ID

479
479
479
479
479
479
479
479
479
110
479
479
120
479
479

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

4790
479
479
479
479
140
479

Defining a Structure

```
struct [structure tag]
{
   member definition;
   member definition;
   ...
   member definition;
```

480
480
480
480
480
480
480
480
480
110
480
480
120
480
480
#include <stdio.h>
4800
480
480
480
480
140
480

```
} [one or more structure variables];
```

To define a structure, you must use the **struct** statement. The struct statement defines a new data type, with more than one member. The format of the struct statement is as follows:The **structure tag** is optional and each member definition is a normal variable definition, such as int i; or float f; or any other valid variable definition. At the end of the structure's definition, before the final semicolon, you can specify one or more structure variables but it is optional. Here is the way you would declare the Book structure:

```
struct Books

{
```

481
481
481
481
481
481
481
481
481
110
481
481
120
481
481

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

4810
481
481
481
481
140
481

```
    char  title[50]; char
          author[50];
    char  subject[100];
```

482
482
482
482
482
482
482
482
482
110
482
482
120
482
482

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

4820
482
482
482
482
140
482

```
    int   book_id;
 } book;
```

Accessing Structure Members

```
/* book 2 specification */
strcpy( Book2.title, "Telecom Billing"); strcpy( Book2.author, "Zara Ali");

/* book 1 specification */
```

483
483
483
483
483
483
483
483
483
110
483
483
120
483
483

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

4830
483
483
483
483
140
483

```
strcpy( Book1.title, "C Programming"); strcpy( Book1.author, "Nuha
Ali");
strcpy( Book1.subject, "C Programming Tutorial");
Book1.book_id = 6495407;
/* Declare Book1 of type Book */
/* Declare Book2 of type Book */
int main( )
{
```

484
484
484
484
484
484
484
484
484
110
484
484
120
484
484

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

4840
484
484
484
484
140
484

```
   struct Books Book1; struct Books Book2;

struct Books

{

   char  title[50]; char      author[50]; char  subject[100]; int book_id;

};
#include <stdio.h>

#include <string.h>
```

To access any member of a structure, we use the **member access operator (.)**. The member access operator is coded as a period between the structure variable name and the structure

485
485
485
485
485
485
485
485
485
110
485
485
120
485
485

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

4850
485
485
485
485
140
485

member that we wish to access. You would use the keyword **struct** to define variables of structure type. The following example shows how to use a structure in a program:

486
486
486
486
486
486
486
486
486
110
486
486
120
486
486
#include <stdio.h>
4860
486
486
486
486
140
486

```
    strcpy( Book2.subject, "Telecom Billing Tutorial");
    Book2.book_id = 6495700;

    /* print Book1 info */
    printf( "Book 1 title : %s\n", Book1.title);
    printf( "Book 1 author : %s\n", Book1.author);
    printf( "Book 1 subject : %s\n", Book1.subject);
```

487
487
487
487
487
487
487
487
487
110
487
487
120
487
487

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

4870
487
487
487
487
140
487

```
    printf( "Book 1 book_id : %d\n", Book1.book_id);

    /* print Book2 info */
    printf( "Book 2 title : %s\n", Book2.title);
    printf( "Book 2 author : %s\n", Book2.author);
    printf( "Book 2 subject : %s\n", Book2.subject);
    printf( "Book 2 book_id : %d\n", Book2.book_id);

    return 0;
```

488
488
488
488
488
488
488
488
488
110
488
488
120
488
488

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

4880
488
488
488
488
140
488

```
}
```

When the above code is compiled and executed, it produces the following result:

ook		title : C Programming
ook		author : Nuha Ali
ook		subject : C Programming Tutorial
ook		book_id : 6495407

489
489
489
489
489
489
489
489
489
110
489
489
120
489
489

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

4890
489
489
489
489
140
489

ook		title : Telecom Billing
ook		author : Zara Ali
ook		subject : Telecom Billing Tutorial
ook		book_id : 6495700

Structures as Function Arguments

490
490
490
490
490
490
490
490
490
110
490
490
120
490
490

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

4900
490
490
490
490
140
490

You can pass a structure as a function argument in the same way as you pass any other variable or pointer.

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

```
491
491
491
491
491
491
491
491
491
110
491
491
120
491
491
#include <stdio.h>
4910
491
491
491
491
140
491
```

```
   struct Books
   {
   char                title[50]; char     author[50]; char        subject[100]; int
book_id;
   };


   /* function declaration */
```

492
492
492
492
492
492
492
492
492
110
492
492
120
492
492

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

4920
492
492
492
492
140
492

```
        void printBook( struct Books book ); int main( )
        {
        struct Books Book1;                         /* Declare Book1 of type Book */ struct
Books Book2;                                        /* Declare Book2 of type Book */

        /* book 1 specification */
        strcpy( Book1.title, "C Programming"); strcpy( Book1.author, "Nuha Ali");
        strcpy( Book1.subject, "C Programming Tutorial"); Book1.book_id = 6495407;

        /* book 2 specification */
        strcpy( Book2.title, "Telecom Billing"); strcpy( Book2.author, "Zara Ali");
```

493
493
493
493
493
493
493
493
493
110
493
493
120
493
493

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

4930
493
493
493
493
140
493

strcpy(Book2.subject, "Telecom Billing Tutorial"); Book2.book_id = 6495700;

/* print Book1 info */ printBook(Book1);

/* Print Book2 info */ printBook(Book2);

494
494
494
494
494
494
494
494
494
110
494
494
120
494
494
#include <stdio.h>
4940
494
494
494
494
140
494

```
   return 0;
}
void printBook( struct Books book )
{
   printf( "Book title : %s\n", book.title);
   printf( "Book author : %s\n", book.author);
   printf( "Book subject : %s\n", book.subject);
```

495
495
495
495
495
495
495
495
495
110
495
495
120
495
495

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

4950
495
495
495
495
140
495

```
   printf( "Book book_id : %d\n", book.book_id);
}
```

When the above code is compiled and executed, it produces the following result:

```
Book title : C Programming
Book author : Nuha Ali
Book subject : C Programming Tutorial
Book book_id : 6495407
Book title : Telecom Billing
```

496
496
496
496
496
496
496
496
496
110
496
496
120
496
496

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

4960
496
496
496
496
140
496

```
Book author : Zara Ali

Book subject : Telecom Billing Tutorial

Book book_id : 6495700
```

Pointers to Structures

```
struct Books *struct_pointer;
```

You can define pointers to structures in the same way as you define pointer to any other variable:Now, you can store the address of a structure variable in the above-defined pointer variable. To find the address of a structure variable, place the '&' operator before the structure's

497
497
497
497
497
497
497
497
110
497
497
120
497
497

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

4970
497
497
497
497
140
497

name as follows:

```
struct_pointer = &Book1;

struct_pointer->title;
```

To access the members of a structure using a pointer to that structure, you must use the -> operator as follows:Let us rewrite the above example using structure pointer.

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

498
498
498
498
498
498
498
498
498
110
498
498
120
498
498

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

4980
498
498
498
498
140
498

```
#include <string.h>

struct Books
{
char title[50]; char author[50]; char subject[100]; int book_id;
};
```

```
499
499
499
499
499
499
499
499
499
110
499
499
120
499
499
#include <stdio.h>
4990
499
499
499
499
140
499
```

```
/* function declaration */
void printBook( struct Books *book ); int main( )
{
struct Books Book1;                    /* Declare Book1 of type Book */ struct
Books Book2;                           /* Declare Book2 of type Book */

/* book 1 specification */
strcpy( Book1.title, "C Programming"); strcpy( Book1.author, "Nuha Ali");
strcpy( Book1.subject, "C Programming Tutorial"); Book1.book_id = 6495407;

/* book 2 specification */
```

500
500
500
500
500
500
500
500
500
110
500
500
120
500
500

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

5000
500
500
500
500
140
500

strcpy(Book2.title, "Telecom Billing"); strcpy(Book2.author, "Zara Ali");
strcpy(Book2.subject, "Telecom Billing Tutorial"); Book2.book_id = 6495700;

/* print Book1 info by passing address of Book1 */ printBook(&Book1);

/* print Book2 info by passing address of Book2 */ printBook(&Book2);

```
    return 0;
```

501
501
501
501
501
501
501
501
501
110
501
501
120
501
501

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

5010
501
501
501
501
140
501

```
 }
 void printBook( struct Books *book )
 {
    printf( "Book title : %s\n", book->title);
    printf( "Book author : %s\n", book->author);
    printf( "Book subject : %s\n", book->subject);
    printf( "Book book_id : %d\n", book->book_id);
 }
```

502
502
502
502
502
502
502
502
502
110
502
502
120
502
502

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

5020
502
502
502
502
140
502

When the above code is compiled and executed, it produces the following result:

```
Book title : C Programming
Book author : Nuha Ali
Book subject : C Programming Tutorial
Book book_id : 6495407
Book title : Telecom Billing
Book author : Zara Ali
Book subject : Telecom Billing Tutorial
```

503
503
503
503
503
503
503
503
503
110
503
503
120
503
503

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

5030
503
503
503
503
140
503

```
Book book_id : 6495700
```

Bit Fields

Bit Fields allow the packing of data in a structure. This is especially useful when memory or data storage is at a premium. Typical examples include:

- Packing several objects into a machine word, e.g. 1 bit flags can be compacted.
- Reading external file formats -- non-standard file formats could be read in, e.g., 9-bit integers.

```
struct packed_struct {
```

504
504
504
504
504
504
504
504
110
504
504
120
504
504

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

5040
504
504
504
504
140
504

```
   unsigned int f1:1;
   unsigned int f2:1;
   unsigned int f3:1;
   unsigned int f4:1;
```

C allows us to do this in a structure definition by putting :bit length after the variable. For example:

```
505
505
505
505
505
505
505
505
505
110
505
505
120
505
505
#include <stdio.h>
5050
505
505
505
505
140
505
```

```
   unsigned int type:4;
   unsigned int my_int:9;
 } pack;
```

Here, the packed_struct contains 6 members: Four 1 bit flags f1..f3, a 4-bit type, and a 9-bit my_int.

C automatically packs the above bit fields as compactly as possible, provided that the maximum length of the field is less than or equal to the integer word length of the computer. If this is not the case, then some compilers may allow memory overlap for the fields, while others would store the next field in the next word.

506
506
506
506
506
506
506
506
506
110
506
506
120
506
506

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

5060
506
506
506
506
140
506

507
507
507
507
507
507
507
507
507
110
507
507
120
507
507
```
#include <stdio.h>
```
5070
507
507
507
507
140
507

Unions

A **union** is a special data type available in C that allows to store different data types in the same memory location. You can define a union with many members, but only one member can contain a value at any given time. Unions provide an efficient way of using the same memory location for multiple purpose.

Defining a Union

To define a union, you must use the **union** statement in the same way as you did while defining a structure. The union statement defines a new data type with more than one member for your program. The format of the union statement is as follows:

508
508
508
508
508
508
508
508
110
508
508
120
508
508
#include <stdio.h>
5080
508
508
508
508
140
508

```
union [union tag]
{
   member definition;
   member definition;
   ...
   member definition;
} [one or more union variables];
union Data
```

509
509
509
509
509
509
509
509
509
110
509
509
120
509
509

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

5090
509
509
509
509
140
509

```
{
   int i;
   float f;
   char  str[20];
} data;
```

The **union tag** is optional and each member definition is a normal variable definition, such as int i; or float f; or any other valid variable definition. At the end of the union's definition, before the final semicolon, you can specify one or more union variables, but it is optional. Here is the way you would define a union type named Data having three members i, f, and str:Now, a

510
510
510
510
510
510
510
510
510
110
510
510
120
510
510

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

5100
510
510
510
510
140
510

variable of **Data** type can store an integer, a floating-point number, or a string of characters. It means a single variable, i.e., same memory location, can

511
511
511
511
511
511
511
511
511
110
511
511
120
511
511
#include <stdio.h>
5110
511
511
511
511
140
511

be used to store multiple types of data. You can use any built-in or user-defined data types inside a union based on your requirement.

The memory occupied by a union will be large enough to hold the largest member of the union. For example, in the above example, Data type will occupy

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

union Data
{
```

512
512
512
512
512
512
512
512
512
110
512
512
120
512
512

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

5120
512
512
512
512
140
512

```
    int i;
    float f;
    char  str[20];
 };

 int main( )
 {
    union Data data;
```

513
513
513
513
513
513
513
513
513
110
513
513
120
513
513

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

5130
513
513
513
513
140
513

```
    printf( "Memory size occupied by data : %d\n", sizeof(data));


    return 0;
 }
```

20 bytes of memory space because this is the maximum space which can be occupied by a character string. The following example displays the total memory size occupied by the above union:When the above code is compiled and executed, it produces the following result:

```
 Memory size occupied by data : 20
```

514
514
514
514
514
514
514
514
514
110
514
514
120
514
514

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

5140
514
514
514
514
140
514

Accessing Union Members

To access any member of a union, we use the **member access operator (.)**. The member access operator is coded as a period between the union variable name and the union member that we wish to access. You would use the keyword **union** to define variables of union type. The following example shows how to use unions in a program:

```
515
515
515
515
515
515
515
515
515
110
515
515
120
515
515
#include <stdio.h>
5150
515
515
515
515
140
515
```

```
#include <string.h>

union Data
{
   int i;
   float f;
   char  str[20];
```

516
516
516
516
516
516
516
516
516
110
516
516
120
516
516

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

5160
516
516
516
516
140
516

```
};

int main( )
{
   union Data data;

   data.i = 10;
   data.f = 220.5;
```

517
517
517
517
517
517
517
517
517
110
517
517
120
517
517
#include <stdio.h>
5170
517
517
517
517
140
517

```
    strcpy( data.str, "C Programming");

    printf( "data.i : %d\n", data.i);
    printf( "data.f : %f\n", data.f);
    printf( "data.str : %s\n", data.str);

    return 0;
 }
```

518
518
518
518
518
518
518
518
518
110
518
518
120
518
518

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

5180
518
518
518
518
140
518

When the above code is compiled and executed, it produces the following result:

```
data.i : 1917853763

data.f : 4122360580327794860452759994368.000000

data.str : C Programming
```

Here, we can see that the values of **i** and **f** members of union got corrupted because the final value assigned to the variable has occupied the memory location and this is the reason that the value of **str** member is getting printed very well.

Now let's look into the same example once again where we will use one variable at a time which is the main purpose of having unions:

```
519
519
519
519
519
519
519
519
519
110
519
519
120
519
519
#include <stdio.h>
5190
519
519
519
519
140
519
```

520
520
520
520
520
520
520
520
520
110
520
520
120
520
520
#include <stdio.h>
5200
520
520
520
520
140
520

```
#include <string.h>

union Data
{
   int i;
   float f;
   char  str[20];
```

521
521
521
521
521
521
521
521
521
110
521
521
120
521
521

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

5210
521
521
521
521
140
521

```
};

int main( )
{
   union Data data;

   data.i = 10;
   printf( "data.i : %d\n", data.i);
```

522
522
522
522
522
522
522
522
522
110
522
522
120
522
522
#include <stdio.h>
5220
522
522
522
522
140
522

```
    data.f = 220.5;
    printf( "data.f : %f\n", data.f);

    strcpy( data.str, "C Programming");
    printf( "data.str : %s\n", data.str);

    return 0;
```

523
523
523
523
523
523
523
523
523
110
523
523
120
523
523

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

5230
523
523
523
523
140
523

```
}
```

When the above code is compiled and executed, it produces the following result:

```
data.i : 10

data.f : 220.500000

data.str : C Programming
```

Here, all the members are getting printed very well because one member is being used at a time.

524
524
524
524
524
524
524
524
524
110
524
524
120
524
524

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

5240
524
524
524
524
140
524

Bit fiels

```
struct
{
  unsigned int widthValidated;
  unsigned int heightValidated;
} status;
```

Suppose your C program contains a number of TRUE/FALSE variables grouped in a

525
525
525
525
525
525
525
525
525
110
525
525
120
525
525

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

5250
525
525
525
525
140
525

structure called status, as follows:This structure requires 8 bytes of memory space but in actual, we are going to store either 0 or 1 in each of the variables. The C programming language offers a better way to utilize the memory space in such situations.

```
struct
{
  unsigned int widthValidated : 1;
  unsigned int heightValidated : 1;
} status;
```

If you are using such variables inside a structure, then you can define the width of a

526
526
526
526
526
526
526
526
110
526
526
120
526
526

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

5260
526
526
526
526
140
526

variable which tells the C compiler that you are going to use only those number of bytes. For example, the above structure can be rewritten as follows:The above structure requires 4 bytes of memory space for status variable, but only 2 bits will be used to store the values.

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

/* define simple structure */
struct
{
```

527
527
527
527
527
527
527
527
527
110
527
527
120
527
527

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

5270
527
527
527
527
140
527

```
unsigned int widthValidated;
```

If you will use up to 32 variables, each one with a width of 1 bit, then also the status structure will use 4 bytes. However, as soon as you have 33 variables, it will allocate the next slot of the memory and it will start using 8 bytes. Let us check the following example to understand the concept:

```
528
528
528
528
528
528
528
528
528
110
528
528
120
528
528
#include <stdio.h>
5280
528
528
528
528
140
528
```

```
  unsigned int heightValidated;
 } status1;

 /* define a structure with bit fields */ struct
 {
   unsigned int widthValidated : 1;
   unsigned int heightValidated : 1;
```

529
529
529
529
529
529
529
529
529
110
529
529
120
529
529

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

5290
529
529
529
529
140
529

```
} status2;

int main( )
{
   printf( "Memory size occupied by status1 : %d\n", sizeof(status1));
   printf( "Memory size occupied by status2 : %d\n", sizeof(status2));

   return 0;
```

530
530
530
530
530
530
530
530
530
110
530
530
120
530
530

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

5300
530
530
530
530
140
530

```
}
```

When the above code is compiled and executed, it produces the following result:

```
Memory size occupied by status1 : 8
Memory size occupied by status2 : 4
```

Bit Field Declaration

The declaration of a bit-field has the following form inside a structure:

```
struct
```

531
531
531
531
531
531
531
531
531
110
531
531
120
531
531

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

5310
531
531
531
531
140
531

```
{
  type [member_name] : width ;
};
```

The following table describes the variable elements of a bit field:

Elements	Description
type	An integer type that determines how a bit-field's value is

532
532
532
532
532
532
532
532
532
110
532
532
120
532
532

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

5320
532
532
532
532
140
532

533
533
533
533
533
533
533
533
533
110
533
533
120
533
533
```
#include <stdio.h>
```
5330
533
533
533
533
140
533

	interpreted. The type may be int, signed int, or unsigned int.
member_name	The name of the bit-field.
width	The number of bits in the bit-field. The width must be less than or equal to the bit width of the specified type.

```
struct
{
```

534
534
534
534
534
534
534
534
534
110
534
534
120
534
534

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

5340
534
534
534
534
140
534

```
   unsigned int age : 3;
} Age;
```

The variables defined with a predefined width are called **bit fields**. A bit field can hold more than a single bit; for example, if you need a variable to store a value from 0 to 7, then you can define a bit-field with a width of 3 bits as follows:The above structure definition instructs the C compiler that the age variable is going to use only 3 bits to store the value. If you try to use more than 3 bits, then it will not allow you to do so. Let us try the following example:

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

535
535
535
535
535
535
535
535
535
110
535
535
120
535
535

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

5350
535
535
535
535
140
535

```
struct
{
  unsigned int age : 3;
} Age;

int main( )
{
```

536
536
536
536
536
536
536
536
536
110
536
536
120
536
536
#include <stdio.h>
5360
536
536
536
536
140
536

```
    Age.age = 4;
    printf( "Sizeof( Age ) : %d\n", sizeof(Age) );
    printf( "Age.age : %d\n", Age.age );

    Age.age = 7;
    printf( "Age.age : %d\n", Age.age );


    Age.age = 8;
```

537
537
537
537
537
537
537
537
537
110
537
537
120
537
537

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

5370
537
537
537
537
140
537

538
538
538
538
538
538
538
538
538
110
538
538
120
538
538
#include <stdio.h>
5380
538
538
538
538
140
538

```
    printf( "Age.age : %d\n", Age.age );

    return 0;
 }
 Sizeof( Age ) : 4
 Age.age : 4
 Age.age : 7
```

539
539
539
539
539
539
539
539
539
110
539
539
120
539
539

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

5390
539
539
539
539
140
539

```
Age.age : 0
```

When the above code is compiled, it will compile with a warning and when executed, it produces the following result:

```
540
540
540
540
540
540
540
540
540
110
540
540
120
540
540
#include <stdio.h>
5400
540
540
540
540
140
540
```

Typedef

```
typedef unsigned char BYTE;
```

The C programming language provides a keyword called **typedef**, which you can use to give a type, a new name. Following is an example to define a term **BYTE** for one-byte numbers:After this type definition, the identifier BYTE can be used as an abbreviation for the type **unsigned char**, for example:

```
BYTE  b1, b2;
```

```
typedef unsigned char byte;
```

By convention, uppercase letters are used for these definitions to remind the user that the

541
541
541
541
541
541
541
541
541
110
541
541
120
541
541

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

5410
541
541
541
541
140
541

type name is really a symbolic abbreviation, but you can use lowercase, as follows:You can use **typedef** to give a name to your user-defined data types as well. For example, you can use typedef with structure to define a new data type and then use that data type to define structure variables directly as follows:

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

typedef struct Books
{
```

542
542
542
542
542
542
542
542
542
110
542
542
120
542
542

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

5420
542
542
542
542
140
542

```
   char  title[50]; char
         author[50];
   char  subject[100];
   int   book_id;
} Book;

int main( )
{
```

543
543
543
543
543
543
543
543
543
110
543
543
120
543
543

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

5430
543
543
543
543
140
543

```
    Book book;

    strcpy( book.title, "C Programming");
```

544
544
544
544
544
544
544
544
544
110
544
544
120
544
544
#include <stdio.h>
5440
544
544
544
544
140
544

```
    strcpy( book.author, "Nuha Ali");
    strcpy( book.subject, "C Programming Tutorial");
    book.book_id = 6495407;

    printf( "Book title : %s\n", book.title);
    printf( "Book author : %s\n", book.author);
    printf( "Book subject : %s\n", book.subject);
```

545
545
545
545
545
545
545
545
545
110
545
545
120
545
545

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

5450
545
545
545
545
140
545

```
   printf( "Book book_id : %d\n", book.book_id);

   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Book  title : C Programming
Book  author : Nuha Ali
Book  subject : C Programming Tutorial
```

546
546
546
546
546
546
546
546
546
110
546
546
120
546
546

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

5460
546
546
546
546
140
546

```
Book  book_id : 6495407
```

typedef vs #define

#define is a C-directive which is also used to define the aliases for various data types similar to **typedef** but with the following differences:

· **typedef** is limited to giving symbolic names to types only, whereas **#define** can be used to define alias for values as well, e.g., you can define 1 as ONE, etc.

· **typedef** interpretation is performed by the compiler whereas **#define** statements are processed by the preprocessor.

The following example shows how to use #define in a program:

547
547
547
547
547
547
547
547
547
110
547
547
120
547
547

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

5470
547
547
547
547
140
547

```
#include <stdio.h>

#define TRUE  1
#define FALSE 0

int main( )
{
   printf( "Value of TRUE : %d\n", TRUE);
```

548
548
548
548
548
548
548
548
548
110
548
548
120
548
548
#include <stdio.h>
5480
548
548
548
548
140
548

549
549
549
549
549
549
549
549
549
110
549
549
120
549
549
#include <stdio.h>
5490
549
549
549
549
140
549

```
   printf( "Value of FALSE : %d\n", FALSE);

   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Value of TRUE : 1
Value of FALSE : 0
```

550
550
550
550
550
550
550
550
550
110
550
550
120
550
550

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

5500
550
550
550
550
140
550

551
551
551
551
551
551
551
551
551
110
551
551
120
551
551
#include <stdio.h>
5510
551
551
551
551
140
551

Input/output

When we say **Input**, it means to feed some data into a program. An input can be given in the form of a file or from the command line. C programming provides a set of built-in functions to read the given input and feed it to the program as per requirement.

When we say **Output**, it means to display some data on screen, printer, or in any file. C programming provides a set of built-in functions to output the data on the computer screen as well as to save it in text or binary files.

The Standard Files

552
552
552
552
552
552
552
552
552
110
552
552
120
552
552
#include <stdio.h>
5520
552
552
552
552
140
552

C programming treats all the devices as files. So devices such as the display are addressed in the same way as files and the following three files are automatically opened when a program executes to provide access to the keyboard and screen.

Standard File	File Pointer	Device
Standard input	stdin	Keyboard
Standard output	stdout	Screen

553
553
553
553
553
553
553
553
553
110
553
553
120
553
553
#include <stdio.h>
5530
553
553
553
553
140
553

Standard error	stderr	Your screen

The file pointers are the means to access the file for reading and writing purpose. This section explains how to read values from the screen and how to print the result on the screen.

The getchar() and putchar() Functions

The **int getchar(void)** function reads the next available character from the screen and returns it as an integer. This function reads only single character at a time. You can use this method in the loop in case you want to read more than one character from the screen.

The **int putchar(int c)** function puts the passed character on the screen and returns the same character. This function puts only single character at a time. You can use this method in the

554
554
554
554
554
554
554
554
554
110
554
554
120
554
554
#include <stdio.h>
5540
554
554
554
554
140
554

loop in case you want to display more than one character on the screen. Check the following example:

```
555
555
555
555
555
555
555
555
555
110
555
555
120
555
555
#include <stdio.h>
5550
555
555
555
555
140
555
```

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

   printf( "Enter a value :"); c
   = getchar( );
```

556
556
556
556
556
556
556
556
556
110
556
556
120
556
556

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

5560
556
556
556
556
140
556

```
    printf( "\nYou entered: ");
    putchar( c );

    return 0;
 }
 $./a.out
 Enter a value : this is test
```

557
557
557
557
557
557
557
557
557
110
557
557
120
557
557

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

5570
557
557
557
557
140
557

```
You entered: t
```

When the above code is compiled and executed, it waits for you to input some text. When you enter a text and press enter, then the program proceeds and reads only a single character and displays it as follows:

The gets() and puts() Functions

The **char *gets(char *s)** function reads a line from **stdin** into the buffer pointed to by **s** until either a terminating newline or EOF (End of File).

```
#include <stdio.h>

int main( )
```

558
558
558
558
558
558
558
558
558
110
558
558
120
558
558

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

5580
558
558
558
558
140
558

```
 {
    char str[100];


    printf( "Enter a value :");
    gets( str );

    printf( "\nYou entered: ");
```

The **int puts(const char *s)** function writes the string ‘s’ and ‘a’ trailing newline to **stdout**.

559
559
559
559
559
559
559
559
559
110
559
559
120
559
559

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

5590
559
559
559
559
140
559

560
560
560
560
560
560
560
560
560
110
560
560
120
560
560
#include <stdio.h>
5600
560
560
560
560
140
560

```
    puts( str );

    return 0;
 }
 $./a.out
 Enter a value : this is test
 You entered: This is test
```

561
561
561
561
561
561
561
561
561
110
561
561
120
561
561

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

5610
561
561
561
561
140
561

When the above code is compiled and executed, it waits for you to input some text. When you enter a text and press enter, then the program proceeds and reads the complete line till end, and displays it as follows:

The scanf() and printf() Functions

The **int scanf(const char *format, ...)** function reads the input from the standard input stream **stdin** and scans that input according to the **format** provided.

The **int printf(const char *format, ...)** function writes the output to the standard output stream **stdout** and produces the output according to the format provided.

The **format** can be a simple constant string, but you can specify %s, %d, %c,

562
562
562
562
562
562
562
562
562
110
562
562
120
562
562
#include <stdio.h>
5620
562
562
562
562
140
562

```
#include <stdio.h>
int main( )
{
   char str[100];
   int i;

   printf( "Enter a value :");
   scanf("%s %d", str, &i);
```

563
563
563
563
563
563
563
563
563
110
563
563
120
563
563

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

5630
563
563
563
563
140
563

```
    printf( "\nYou entered: %s %d ", str, i);


    return 0;
```

%f, etc., to print or read strings, integer, character, or float, respectively. There are many other formatting options available which can be used based on requirements. Let us now proceed with a simple example to understand the concepts better:

564
564
564
564
564
564
564
564
564
110
564
564
120
564
564

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

5640
564
564
564
564
140
564

```
}
$./a.out
Enter a value : seven 7
You entered: seven 7
```

When the above code is compiled and executed, it waits for you to input some text. When you enter a text and press enter, then program proceeds and reads the input and displays it as follows:Here, it should be noted that scanf() expects input in the same format as you provided %s and %d, which means you have to provide valid inputs like "string integer". If you provide "string

565
565
565
565
565
565
565
565
565
110
565
565
120
565
565

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

5650
565
565
565
565
140
565

string" or "integer integer", then it will be assumed as wrong input. Secondly, while reading a string, scanf() stops reading as soon as it encounters a space, so "this is test" are three strings for scanf().

566
566
566
566
566
566
566
566
566
110
566
566
120
566
566
#include <stdio.h>
5660
566
566
566
566
140
566

File

The last chapter explained the standard input and output devices handled by C programming language. This chapter covers how C programmers can create, open, close text or binary files for their data storage.

A file represents a sequence of bytes, regardless of it being a text file or a binary file. C programming language provides access on high-level functions as well as low-level (OS level) calls to handle file on your storage devices. This chapter will take you through the important calls for file management.

Opening Files

567
567
567
567
567
567
567
567
567
110
567
567
120
567
567
#include <stdio.h>
5670
567
567
567
567
140
567

```
FILE *fopen( const char * filename, const char * mode );
```

You can use the **fopen( )** function to create a new file or to open an existing file. This call will initialize an object of the type **FILE**, which contains all the information necessary to control the stream. The prototype of this function call is as follows:Here, **filename** is a string literal, which you will use to name your file, and access **mode** can have one of the following values:

Mode	Description
r	Opens an existing text file for reading purpose.

568
568
568
568
568
568
568
568
568
110
568
568
120
568
568

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

5680
568
568
568
568
140
568

w	Opens a text file for writing. If it does not exist, then a new file is created. Here your program will start writing content from the beginning of the file.
a	Opens a text file for writing in appending mode. If it does not exist, then a new file is created. Here your program will start appending content in the existing file content.
r+	Opens a text file for both reading and writing.
w+	Opens a text file for both reading and writing. It first truncates the file to

569
569
569
569
569
569
569
569
569
110
569
569
120
569
569

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

5690
569
569
569
569
140
569

	zero length if it exists, otherwise creates a file if it does not exist.

570
570
570
570
570
570
570
570
570
110
570
570
120
570
570

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

5700
570
570
570
570
140
570

Opens a text file for both reading and writing. It creates the file if it does not exist. The reading will start from the beginning but writing can only be appended.

```
"rb", "wb", "ab", "rb+", "r+b", "wb+", "w+b", "ab+", "a+b"
```

a+

If you are going to handle binary files, then you will use the following access modes instead of the above-mentioned ones:

Closing a File

571
571
571
571
571
571
571
571
571
110
571
571
120
571
571

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

5710
571
571
571
571
140
571

To close a file, use the fclose( ) function. The prototype of this function is:

```
int fclose( FILE *fp );
```

The **fclose()** function returns zero on success, or **EOF** if there is an error in closing the file. This function actually flushes any data still pending in the buffer to the file, closes the file, and releases any memory used for the file. The EOF is a constant defined in the header file **stdio.h**.

There are various functions provided by C standard library to read and write a file, character by character, or in the form of a fixed length string.

Writing a File

572
572
572
572
572
572
572
572
572
110
572
572
120
572
572
#include <stdio.h>
5720
572
572
572
572
140
572

Following is the simplest function to write individual characters to a stream:

```
int fputc( int c, FILE *fp );
```

The function **fputc()** writes the character value of the argument c to the output stream referenced by fp. It returns the written character written on success otherwise **EOF** if there is an error. You can use the following functions to write a null-terminated string to a stream:

```
int fputs( const char *s, FILE *fp );
```

The function **fputs()** writes the string **s** to the output stream referenced by fp. It returns a non-negative value on success, otherwise **EOF** is returned in case of any error. You can use **int fprintf(FILE *fp,const char *format, ...)** function as well to write a string into a file. Try the following example.

Make sure you have **/tmp** directory available. If it is not, then before proceeding, you

573
573
573
573
573
573
573
573
573
110
573
573
120
573
573

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

5730
573
573
573
573
140
573

must create this directory on your machine.

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

```
574
574
574
574
574
574
574
574
574
110
574
574
120
574
574
#include <stdio.h>
5740
574
574
574
574
140
574
```

```
 main()
 {
    FILE *fp;

    fp = fopen("/tmp/test.txt", "w+");
    fprintf(fp, "This is testing for fprintf...\n");
```

575
575
575
575
575
575
575
575
575
110
575
575
120
575
575
#include <stdio.h>
5750
575
575
575
575
140
575

```
    fputs("This is testing for fputs...\n", fp);
    fclose(fp);
 }
```

When the above code is compiled and executed, it creates a new file **test.txt** in /tmp directory and writes two lines using two different functions. Let us read this file in the next section.

Reading a File

Given below is the simplest function to read a single character from a file:

576
576
576
576
576
576
576
576
576
110
576
576
120
576
576
#include <stdio.h>
5760
576
576
576
576
140
576
576

```
int fgetc( FILE * fp );
```

The **fgetc()** function reads a character from the input file referenced by fp. The return value is the character read, or in case of any error, it returns **EOF**. The following function allows to read a string from a stream:

```
char *fgets( char *buf, int n, FILE *fp );
```

The functions **fgets()** reads up to n - 1 characters from the input stream referenced by fp. It copies the read string into the buffer **buf**, appending a **null** character to terminate the string.

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

577
577
577
577
577
577
577
577
577
110
577
577
120
577
577

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

5770
577
577
577
577
140
577

```
main()
{
   FILE *fp;
   char buff[255];
```

If this function encounters a newline character '\n' or the end of the file EOF before they have read the maximum number of characters, then it returns only the characters read up to that point including the new line character. You can also use **int fscanf(FILE *fp, const char *format, ...)** function to read strings from a file, but it stops reading after encountering the first space character.

578
578
578
578
578
578
578
578
578
110
578
578
120
578
578

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

5780
578
578
578
578
140
578

579
579
579
579
579
579
579
579
579
110
579
579
120
579
579
```
#include <stdio.h>
```
5790
579
579
579
579
140
579

```
    fp = fopen("/tmp/test.txt", "r");
    fscanf(fp, "%s", buff);
    printf("1 : %s\n", buff );

    fgets(buff, 255, (FILE*)fp);
    printf("2: %s\n", buff );
```

580
580
580
580
580
580
580
580
580
110
580
580
120
580
580

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

5800
580
580
580
580
140
580

```
    fgets(buff, 255, (FILE*)fp);
    printf("3: %s\n", buff );
    fclose(fp);

 }
 1 : This
 2: is testing for fprintf...

581
581
581
581
581
581
581
581
581
110
581
581
120
581
581
#include <stdio.h>
5810
581
581
581
581
140
581
```

```
3: This is testing for fputs...
```

When the above code is compiled and executed, it reads the file created in the previous section and produces the following result:Let's see a little more in detail about what happened here. First, **fscanf()** reads just **This** because after that, it encountered a space, second call is for **fgets()** which reads the remaining line till it encountered end of line. Finally, the last call **fgets()** reads the second line completely.

```
size_t fread(void *ptr, size_t size_of_elements, size_t
             number_of_elements, FILE *a_file);
```

582
582
582
582
582
582
582
582
582
110
582
582
120
582
582

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

5820
582
582
582
582
140
582

```
size_t fwrite(const void *ptr, size_t size_of_elements,
              size_t number_of_elements, FILE *a_file);
```

Binary I/O Functions There are two functions that can be used for binary input and output:

Both of these functions should be used to read or write blocks of memories - usually arrays or structures.

583
583
583
583
583
583
583
583
583
110
583
583
120
583
583

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

5830
583
583
583
583
140
583

Literature

1. Learn C programming language/ tutorials ,185, 2014
http://www.kciti.edu/wp-content/uploads/2017/07/cprogramming_tutorial.pdf
2.

584
584
584
584
584
584
584
584
584
110
584
584
120
584
584
#include <stdio.h>
5840
584
584
584
584
140
584