BCA 1ST Semester

BCA - 101: COMPUTER FUNDAMENTALS AND PROGRAMMING

UNIT: 4

(Programming Using C)

C programming :

C programming is a general-purpose, procedural, imperative computer programming language developed in 1972 by Dennis M. Ritchie at the Bell Telephone Laboratories. C is the most widely used computer language. It keeps fluctuating at number one scale of popularity along with Java programming language, which is also equally popular and most widely used among modern software programmers.

Tokens in C

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 –

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

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).

```
mohd       zara    abc   move_name  a_123
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
case	extern	return	union

char	float	short	unsigned
const	for	signed	void
continue	goto	sizeof	volatile
default	if	static	while
do	int	struct	_Packed

Data types :

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.

Integer Types

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

Type	Storage size	Value range
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
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	8 bytes or (4bytes for 32 bit OS)	-9223372036854775808 to 9223372036854775807
unsigned long	8 bytes	0 to 18446744073709551615

Floating-Point Types

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

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

The void Type

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

Sr.No.	Types & 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);**
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.

Variable :

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.

```
type variable_list;
```

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 −

Sr.No.	Type & Description
1	**char** Typically a single octet(one byte). It is an integer type.
2	**int** The most natural size of integer for the machine.
3	**float** A single-precision floating point value.
4	**double** A double-precision floating point value.
5	**void** Represents the absence of type.

Constant :

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

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.

Here are some examples of integer literals −

```
212         /* Legal */
215u        /* Legal */
0xFeeL      /* Legal */
078         /* Illegal: 8 is not an octal digit */
032UU       /* Illegal: cannot repeat a suffix */
```

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 −

```
3.14159       /* Legal */
314159E-5L    /* Legal */
510E          /* Illegal: incomplete exponent */
210f          /* Illegal: no decimal or exponent */
.e55          /* Illegal: missing integer or fraction */
```

Character Constants

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).

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.

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

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

```
"hello, dear"

"hello, \

dear"

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

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.

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
−	Subtracts second operand from the first.	A − B = -10
*	Multiplies both operands.	A * B = 200
/	Divides numerator by de-numerator.	B / A = 2
%	Modulus Operator and remainder of after an integer division.	B % A = 0
++	Increment operator increases the integer value by one.	A++ = 11
--	Decrement operator decreases the integer value by one.	A-- = 9

```c
#include <stdio.h>

main() {

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

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

```
Line 1 - Value of c is 31
Line 2 - Value of c is 11
Line 3 - Value of c is 210
Line 4 - Value of c is 2
Line 5 - Value of c is 1
Line 6 - Value of c is 21
Line 7 - Value of c is 22
```

Relational Operators :

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

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

```c
#include <stdio.h>

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" );
   }

   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 ) {
      printf("Line 3 - a is greater than b\n" );
   } else {
      printf("Line 3 - a is not greater than b\n" );
   }
```

```
   /* Lets change value of a and b */
   a = 5;
   b = 20;

   if ( a <= b ) {
      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" );
   }
}
```

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

```
Line 1 - a is not equal to b
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
```

Logical Operators

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

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

```
#include <stdio.h>

main() {

   int a = 5;
   int b = 20;
   int c ;

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

   if ( a || b ) {
      printf("Line 2 - Condition is true\n" );
   }
```

```
/* lets change the value of  a and b */
a = 0;
b = 10;

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

if ( !(a && b) ) {
   printf("Line 4 - Condition is true\n" );
}
```

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

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

Bitwise Operators

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

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

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

A = 0011 1100

B = 0000 1101

A&B = 0000 1100

A|B = 0011 1101

A^B = 0011 0001

~A = 1100 0011

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

```
#include <stdio.h>

main() {

   unsigned int a = 60;        /* 60 = 0011 1100 */
   unsigned int b = 13;        /* 13 = 0000 1101 */
   int c = 0;

   c = a & b;        /* 12 = 0000 1100 */
   printf("Line 1 - Value of c is %d\n", c );

   c = a | b;        /* 61 = 0011 1101 */
   printf("Line 2 - Value of c is %d\n", c );

   c = a ^ b;        /* 49 = 0011 0001 */
   printf("Line 3 - Value of c is %d\n", c );

   c = ~a;           /*-61 = 1100 0011 */
   printf("Line 4 - Value of c is %d\n", c );

   c = a << 2;     /* 240 = 1111 0000 */
   printf("Line 5 - Value of c is %d\n", c );

   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 −

```
Line 1 - Value of c is 12
Line 2 - Value of c is 61
Line 3 - Value of c is 49
Line 4 - Value of c is -61
Line 5 - Value of c is 240
Line 6 - Value of c is 15
```

Assignment Operators :

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

Operator	Description	Example
=	Simple assignment operator. Assigns values from right side operands to left side operand	C = A + B will assign the value of A + B

		to C
+=	Add AND assignment operator. It adds the right operand to the left operand and assign the result to the left operand.	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator. It subtracts the right operand from the left operand and assigns the result to the left operand.	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator. It multiplies the right operand with the left operand and assigns the result to the left operand.	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator. It divides the left operand with the right operand and assigns the result to the left operand.	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator. It takes modulus using two operands and assigns the result to the left operand.	C %= A is equivalent to C = C % A
<<=	Left shift AND assignment operator.	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator.	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator.	C &= 2 is same as C = C & 2
^=	Bitwise exclusive OR and assignment operator.	C ^= 2 is same as C = C ^ 2

|=	Bitwise inclusive OR and assignment operator.	C |= 2 is same as C = C | 2

Example

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

Live Demo

```
#include <stdio.h>

main() {

   int a = 21;
   int c ;

   c =  a;
   printf("Line 1 - =  Operator Example, Value of c = %d\n", c );

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

   c -=  a;
   printf("Line 3 - -= Operator Example, Value of c = %d\n", c );

   c *=  a;
   printf("Line 4 - *= Operator Example, Value of c = %d\n", c );

   c /=  a;
   printf("Line 5 - /= Operator Example, Value of c = %d\n", c );

   c  = 200;
   c %=  a;
   printf("Line 6 - %= Operator Example, Value of c = %d\n", c );

   c <<=  2;
   printf("Line 7 - <<= Operator Example, Value of c = %d\n", c );

   c >>=  2;
   printf("Line 8 - >>= Operator Example, Value of c = %d\n", c );

   c &=  2;
   printf("Line 9 - &= Operator Example, Value of c = %d\n", c );

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

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

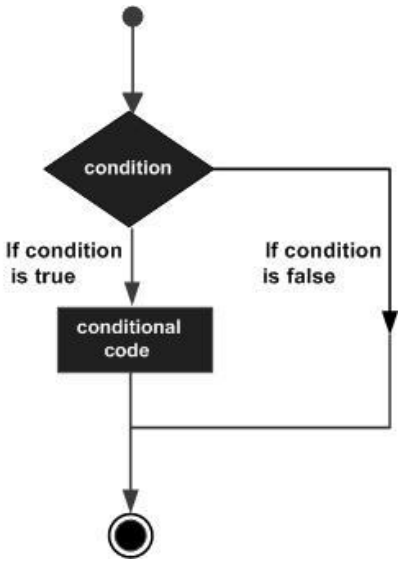
When you compile and execute the above program, it produces the following result −

```
Line 1 - =  Operator Example, Value of c = 21
Line 2 - += Operator Example, Value of c = 42
Line 3 - -= Operator Example, Value of c = 21
Line 4 - *= Operator Example, Value of c = 441
Line 5 - /= Operator Example, Value of c = 21
Line 6 - %= Operator Example, Value of c = 11
Line 7 - <<= Operator Example, Value of c = 44
Line 8 - >>= Operator Example, Value of c = 11
Line 9 - &= Operator Example, Value of c = 2
Line 10 - ^= Operator Example, Value of c = 0
Line 11 - |= Operator Example, Value of c = 2
```

Decision making structures :

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.

Show below is the general form of a typical decision making structure found in most of the programming languages –



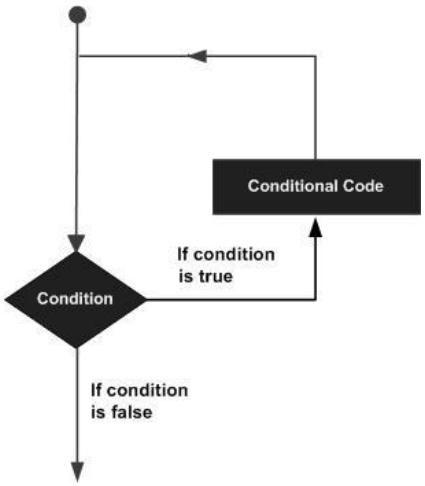
C programming language assumes any **non-zero** and **non-null** values as **true**, and if it is either **zero** or **null**, then it is assumed as **false** value.

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

Sr.No.	Statement & Description
1	if statement An **if statement** consists of a boolean expression followed by one or more statements.
2	if...else statement An **if statement** can be followed by an optional **else statement**, which executes when the Boolean expression is false.
3	nested if statements You can use one **if** or **else if** statement inside another **if** or **else if** statement(s).
4	switch statement A **switch** statement allows a variable to be tested for equality against a list of values.
5	nested switch statements You can use one **switch** statement inside another **switch** statement(s).

Loop :

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 requirements.

Sr.No.	Loop Type & Description
1	while loop Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.
2	for loop Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.
3	do...while loop It is more like a while statement, except that it tests the condition at the end of the loop body.
4	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 condition is true.

Syntax

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

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

for loop :

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);   }
```

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

nested loops :

C programming allows to use one loop inside another loop. The following section shows a 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);
   }
   statement(s);
}
```

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

```
while(condition) {

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

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

```
do {
   statement(s);

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

}while( condition );
```