Operators, Expressions, Precedence and Associativity, Expression Evaluation, Type conversions.

OPERATORS

An operator is a symbol that tells the compiler to perform specific mathematical or logical functions.

The operators are classified into three types based on the number of operands it operates.

1. Unary operator: If operator operates on single operand.

2. Binary operator: If operator operates on two operands.

3. Ternary operator: If operator operates on three operands.

C language is rich in built-in operators .C language provides eight different types of operators. They are

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Assignment Operators
- Increment and decrement Operators
- Bitwise Operators
- conditional operators
- special Operators

Arithmetic Operators

The various arithmetic operators supported by C language are shown in the following table.

Assume variable **A** holds 10 and variable **B** holds 20 then −

Operator	Type	Description	Example
+	Unary	Indicates positive number	+ A=10
	Binary	Adds two operands.	A + B = 30
–	Unary	Indicates negative number	-A=-10
	Binary	Subtracts second operand from the first.	A − B = -10
*	Binary	Multiplies both operands.	A * B = 200
/	Binary	Divides numerator by de-numerator.	B / A = 2
%	Binary	Modulus Operator and remainder of after an integer division.	B % A = 0

Arithmetic operators except Modulus Operator work on integers and float. Modulus operator works only on integers.

Ex -7%3= -1

7%-3=1

- 7%-3=-1

Ex: 3.2%2 is invalid

Relational Operators

Relational operators are used to compare two values. C language supports six types of relational operator and all are binary types. 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. so 0 is returned
!=	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. So 1 is returned.
>	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. So 1 is returned
<	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. So 1 is returned
>=	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. so 0 is returned
<=	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. So 1 is returned

Relational operators works on all kinds of data types.

Logical Operators

Logical operators are used to join two conditions. The logical operators either return true (1) or false (0).C language supports three types of logical operators.

Logical operators work on all data types.

Following table shows the logical operators supported by C language.

Operator	Description	Type
&&	Called Logical AND operator. If both the operands are non-zero, then the condition becomes true.	Binary
||	Called Logical OR Operator. If any of the two operands is non-zero, then the condition becomes true.	Binary
!	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.	Unary

The truth table for logical operators is

Cond-1	Cond-2	Cond-1 && cond-2	Cond-1 || cond-2	! cond-1
F	F	F	F	T
F	T	F	T	T
T	F	F	T	F
T	T	T	T	F

Increment and decrement operators

Increment and decrement operators are used to increase or decrease a value by 1.

These operators are unary operators and **they work only on integer data types**.

Increment operators are of two types pre increment and post increment.

If the operator occurs before operand then the operator is Pre increment operators. Ex ++a

If the operator occurs after operand then the operator is Post increment operators. Ex a++

Decrement operators are of two types pre decrement and post decrement.

If the operator occurs before operand then the operator is Pre decrement operators. Ex --a

If the operator occurs after operand then the operator is Post decrement operators. Ex a--

Operator	Type	Description
++	Unary	Increment operator increases the integer value by one.
--	Unary	Decrement operator decreases the integer value by one.

Ex: If a=5 then a++ will result a=6.

If m=4 then ++m will result m=5.

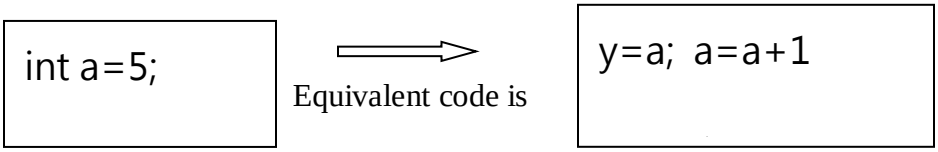
The difference between pre increment and post increment operator lies in the usage of expressions.

Ex:

int a=5;

y=a++;

then value of a and y after execution is a=6,y=5.



Assignment Operators

The assignment operator is used to assign RHS value to LHS variable. The syntax of assignment operator is

var_name=expression;

where expression can be either constant or variable or expression.

Examples of valid assignment statements:

- a=10;
- a=b;
- a=x+y;

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

Operator	Description	Example
+=	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

Assignment operators work on all data types.

Conditional Operator

The symbol for conditional operator is (?:) and is a ternary operator.

The syntax for conditional operator is

cond?expr1:expr2;

If **cond** is true then **expr1** is evaluated otherwise **expr2** is evaluated.

Conditional operator works on all data types.

Bitwise Operators

Bitwise operator works on bits and performs bit-by-bit operation. These operators work on integer data types.

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B)
|	Binary OR Operator copies a bit if it exists in either operand.	(A | B)
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B)
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A)
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2

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 = 60&13=0000 1100=12

A|B =60|13= 0011 1101=61

A^B = 60^13=0011 0001=49

~A = 1100 0011=-61

A<<2=60<<2=0011 1100<<2=1111 0000=240

B>>1=13>>1=0000 1101>>1=0000 0110=6

Special Operators

Operator	Description	Example
sizeof()	Returns the size of a variable in bytes.	sizeof(a), where a is integer, will return 4.

&	Returns the address of a variable.	&a; returns the actual address of the variable.
,	Comma Operator	int a,b;

Ex: z=(x=2,y=3,x+y);

z=5;

Operators Precedence in C

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.

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**
Parenthesis	() []	Left to right
Unary	(postincrement) ++ (postdecrement) - -	Right to left
Unary	+ - ! ~ (preincrement)++ predecrement- - (type)* & sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right

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

Type Conversion in C

When variables and constants of different types are combined in an expression then they are converted to same data type.

The process of converting one predefined type into another is called type conversion.

Type conversion in **C** can be classified into the following two types:

1. Implicit Type Conversion

2. Explicit Type Conversion

Implicit Type Conversion

When the type conversion is performed automatically by the compiler without programmers intervention, such type of conversion is known as **implicit type conversion** or **type promotion**.

The compiler converts all operands into the data type of the largest operand.

The sequence of rules that are applied while evaluating expressions are given below:

All short and char are automatically converted to int, then,

1. If either of the operand is of type long double, then others will be converted to long double and result will be long double.
2. Else, if either of the operand is double, then others are converted to double.
3. Else, if either of the operand is float, then others are converted to float.
4. Else, if either of the operand is unsigned long int, then others will be converted to unsigned long int.
5. Else, if one of the operand is long int, and the other is unsigned int, then
    1. if a long int can represent all values of an unsigned int, the unsigned int is converted to long int.
    2. otherwise, both operands are converted to unsigned long int.
6. Else, if either operand is long int then other will be converted to long int.
7. Else, if either operand is unsigned int then others will be converted to unsigned int.

It should be noted that the final result of expression is converted to type of variable on left side of assignment operator before assigning value to it.

Also, conversion of float to int causes truncation of fractional part, conversion of double to float causes rounding of digits and the conversion of long int to int causes dropping of excess higher order bits.

Explicit Type Conversion

The type conversion performed by the programmer by posing the data type of the expression of specific type is known as explicit type conversion.

The explicit type conversion is also known as **type casting**.

Type casting in **C** is done in the following form:

(data_type)expression;

where, *data_type* is any valid c data type, and *expression* may be constant, variable or expression.

For example,

```
x=(int)a+b*d;
```

The following rules have to be followed while converting the expression from one type to another to avoid the loss of information:

1. All integer types to be converted to float.
2. All float types to be converted to double.
3. All character types to be converted to integer.

Unit II(part 2)

Repetition statements (loops)-while, for, do-while statements, Nested Loops.

Unconditional statements-break, continue, goto.

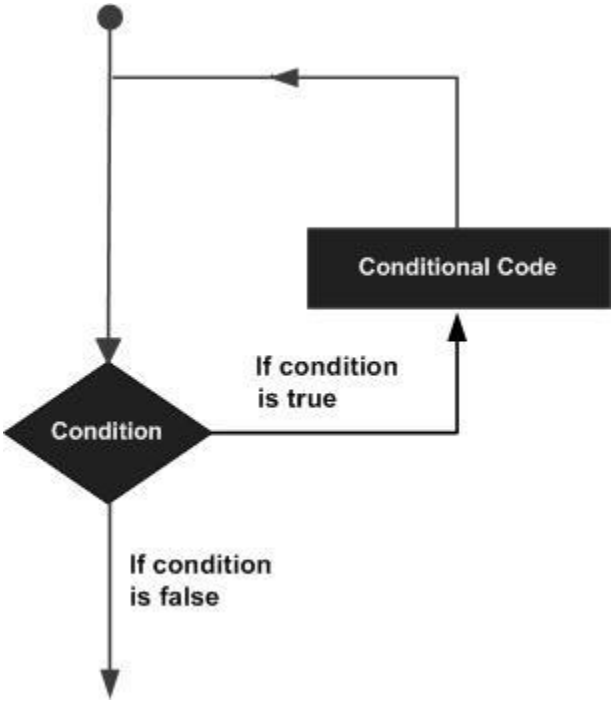
Pointers – Pointer variable, pointer declaration, Initialization of pointer, Accessing variables through pointers, pointers to pointers, pointers to void.

LOOPS

We 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 −



Loops in C Programming can be divided into two types

1.Entry controlled loops: The types of loop where the test condition is stated before the body of the loop, are known as the entry controlled loop. So in the case of an entry controlled loop, the

condition is tested before the execution of the loop. If the test condition is true, then the loop gets the execution, otherwise not.

Example:while,for

2.Exit controlled loops: The types of loop where the test condition is stated at the end of the body of the loop, are know as the exit controlled loops. So, in the case of the exit controlled loops, the body of the loop gets execution without testing the given condition for the first time. Then the condition is tested. If it comes true, then the loop gets another execution and continues till the result of the test condition is not false.

Example :do-while

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

S.N.	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

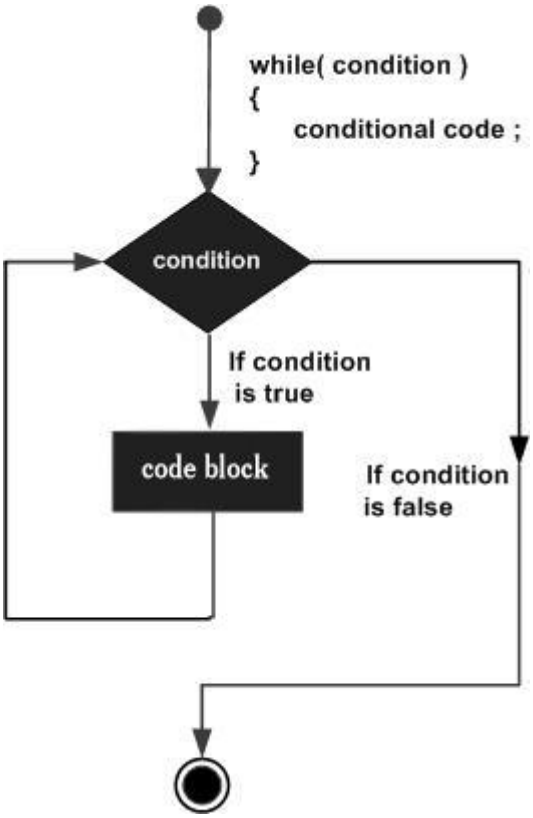
The syntax of a **while** loop in C programming language is −

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

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 true.

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

Flow Diagram



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.

Example

```
#include <stdio.h>

int main () {

   /* local variable definition */
   int a = 10;

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

   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
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

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.

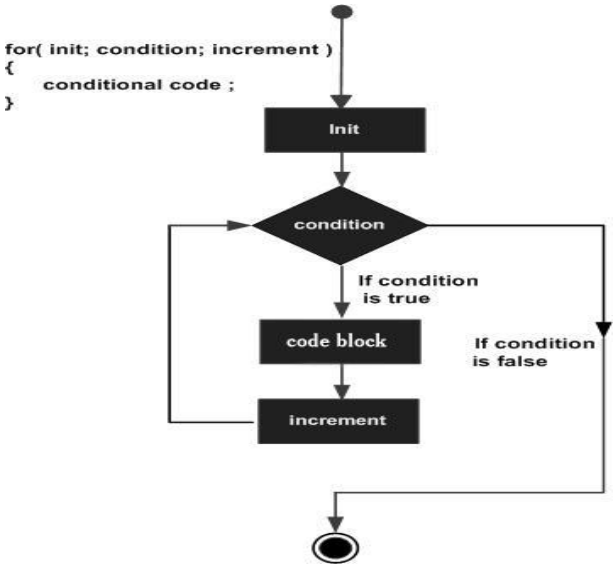
Syntax

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 −

- 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.
- 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.
- 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.
- 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.



Example

```
#include <stdio.h>

int main () {

   int a;

   /* for loop execution */
   for( a = 10; a < 20; a = a + 1 ){
      printf("value of a: %d\n", a);
   }

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

do…while

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

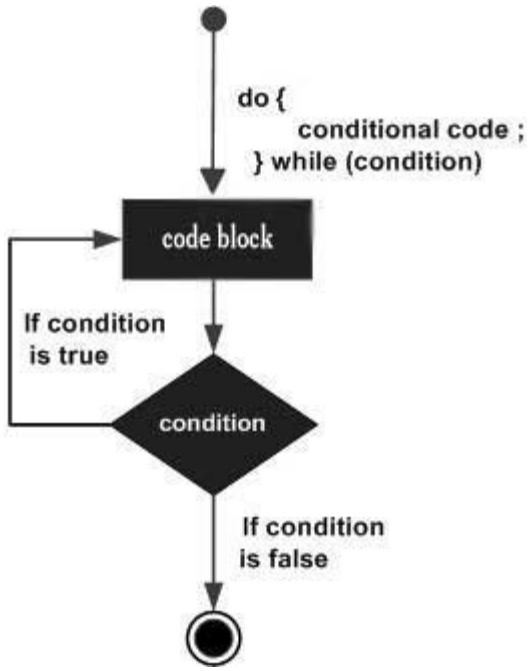
The syntax of a **do...while** loop in C programming language is −

```
do {
   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.

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



Example

```
#include <stdio.h>

int main () {

   /* local variable definition */
   int a = 10;

   /* do loop execution */
   do {
      printf("value of a: %d\n", a);
```

```
    a = a + 1;
  }while( a < 20 );

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

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 );
```

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

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

```
#include <stdio.h>

int main () {

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

   for(i = 2; i<20; i++) {

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

   return 0;
}
```

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

```
2 is prime
3 is prime
5 is prime
```

```
7 is prime
11 is prime
13 is prime
17 is prime
19 is prime
```

Unconditional Statements in C: There are three types of unconditional statements

1.**goto** statement 2.**break** **3.continue**

1) **goto statement : In C programming, goto statement is used for altering the normal sequence of program execution by transferring control to some other part of the program.**

syntax:

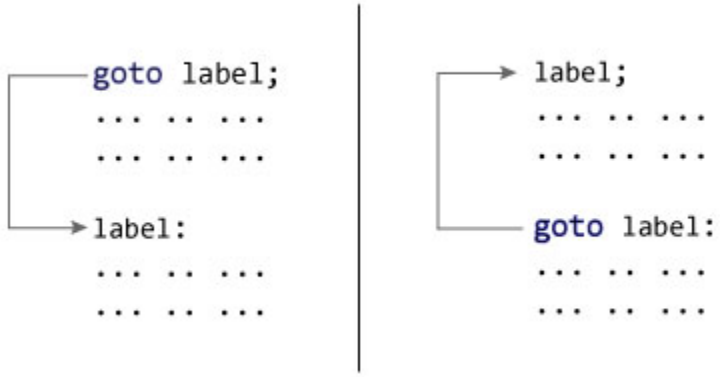
goto label;

……..

……..

label: statement;

In this syntax, **label** is an identifier.
When, the control of program reaches to goto statement, the control of the program will jump to the **label:** and executes the code below it.



Note:

Though goto statement is included in ANSI standard of C, use of goto statement should be reduced as much as possible in a program.

Reasons to avoid goto statement

- Though, using goto statement give power to jump to any part of program, using goto statement makes the logic of the program complex and tangled.
- In modern programming, goto statement is considered a harmful construct and a bad programming practice.
- The goto statement can be replaced in most of C program with the use of break and continue statements.
- In fact, any program in C programming can be perfectly written without the use of goto statement.
- All programmer should try to avoid goto statement as possible as they can.

C supports the following control statements.

S.N.	Control Statement & Description
1	**break statement** Terminates the **loop** or **switch** statement and transfers execution to the statement immediately following the loop or switch.
2	**continue statement** Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.

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.

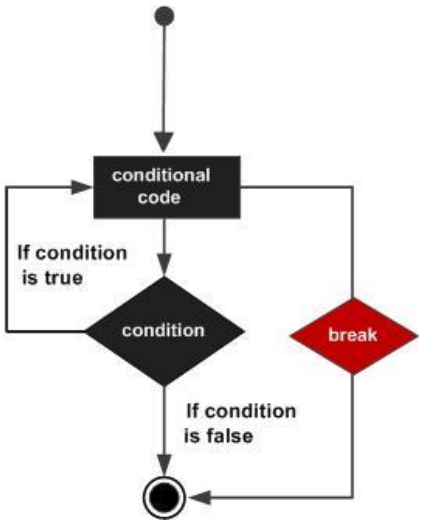
If you are using nested loops, the break statement will stop the execution of the innermost loop and start executing the next line of code after the block.

Syntax

The syntax for a **break** statement in C is as follows −

```
break;
```

Flow Diagram



Example

```
#include <stdio.h>

int main () {

   int a = 10;

  while( a < 20 ) {

   printf("value of a: %d\n", a);
   a++;
   if( a > 15)
   break;

  }
}
```

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
value of a: 15
```

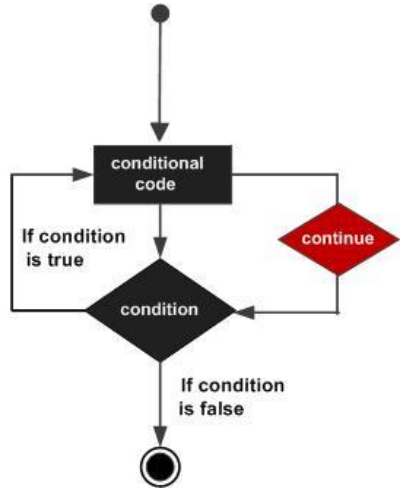
The **continue** statement in C programming works somewhat like the **break** statement. Instead of forcing termination, it forces the next iteration of the loop to take place, skipping any code in between.

Syntax

The syntax for a **continue** statement in C is as follows −

```
continue;
```

Flow Diagram



Example

```
#include <stdio.h>

int main () {

    int a = 10;

    do {

    if( a == 15) {
       a = a + 1;
      continue;
    }
    printf("value of a: %d\n", a);
    a++;
     } while( a < 20 );
}
```

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

POINTERS: Pointer is an entity which contains a memory address.

Definition:

Pointer is a variable that hold address of another variable of same data type.

Uses of Pointers:

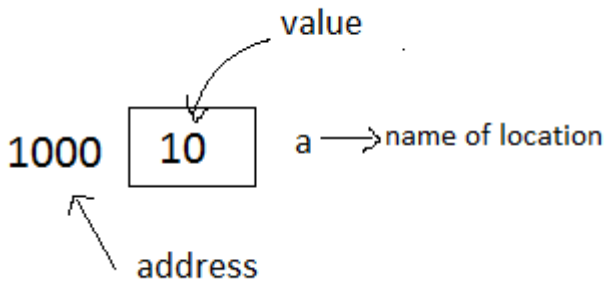
- Accessing array elements
- Passing arguments to functions by reference(i.e arguments can be modified)
- Passing arrays and strings to functions
- Creating data structures such as linked lists, trees, graphs, and so on.
- Obtaining memory from the system dynamically
- It reduces length and the program execution time.

Concept of Pointer:

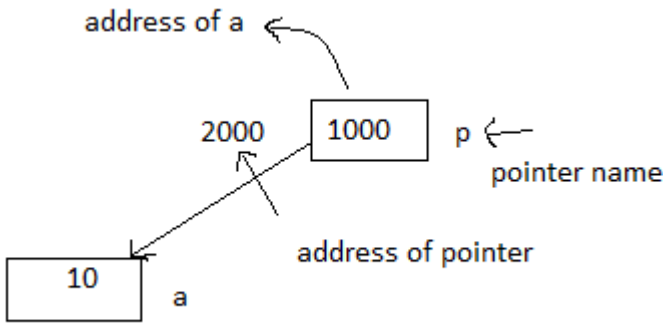
Whenever a **variable** is declared, system will allocate a location to that variable in the memory, to hold value. This location will have its own address number.

Let us assume that system has allocated memory location 1000 for a variable **a**.

int a = 10 ;



We can access the value 10 by either using the variable name **a** or the address 1000. Since the memory addresses are simply numbers they can be assigned to some other variable. The variable that holds memory addresses are called **pointer variables**. A **pointer** variable is therefore nothing but a variable that contains an address, which is a location of another variable. Value of **pointer variable** will be stored in another memory location.



Declaring a pointer variable:

General syntax of pointer declaration is,

data-type *pointer_name;

Data type of pointer must be same as the variable, which the pointer is pointing.

Initialization of Pointer variable:

Pointer initialization is the process of assigning address of a variable to **pointer** variable. Pointer variable contains address of variable of same data type. In C language **address operator** & is used to determine the address of a variable. The & (immediately preceding a variable name) returns the address of the variable associated with it.

```
int a = 10 ;
int *ptr ;          //pointer declaration
ptr = &a ;         //pointer initialization
(or)
int *ptr = &a ;  //initialization and declaration together
```

Pntr1.c

```
#include<stdio.h>
int main()
{
int a = 10;
int *ptr;
ptr = &a;
printf("\nValue of a=%d and &a=%u",a,&a);
printf("\nValue of ptr=%u and *ptr=%d",ptr,*ptr);
return(0);
}
```

Output:

```
Value of a=10 and &a=4356
Value of ptr=4356 and *ptr=10
```

Pointer operations:

The following operators are valid on pointer variables

1. Increment and decrement operators

2. Assignment operators

3. Equal to and not equal to operators

1. Increment and decrement operators:

when increment operator is applied on pointer variable then it is incremented by scale factor (number of bytes required by the data type)of the pointer datatype

To understand pointer arithmetic, let us consider that **ptr** is an integer pointer which points to the address 1000. Assuming 16-bit integers, let us perform the following arithmetic operation on the pointer –

ptr++;

After the above operation, the **ptr** will point to the location 1002 because each time ptr is incremented, it will point to the next integer location which is 2 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.

ptr++ is equivalent to ptr + (sizeof(pointer_data_type)).

- A pointer to an int accesses 2 consecutive bytes of memory
- A pointer to a char accesses 1 consecutive byte of memory
- A pointer to a float accesses 4 consecutive bytes of memory
- A pointer to a double accesses 8 consecutive bytes of memory

The C language allows arithmetic operations to be performed on pointer variables. It is, however, the responsibility of the programmer to see that the result obtained by performing pointer arithmetic is the address of relevant and meaningful data.

```
#include<stdio.h>
main()
{
   int i=3,*x;
   float j=1.5,*y;
   char k='c',*z;
   printf("Value of i=%d\n",i);
   printf("Value of j=%.2f\n",j);
   printf("Value of k=%c\n",k);
   x=&i;
   y=&j;
   z=&k;
   printf("Original value in x=%u\n",x);
   printf("Original value in y=%u\n",y);
   printf("Original value in z=%u\n",z);
   x++;
   y++;
   z++;
   printf("New value in x=%u\n",x);
   printf("New value in y=%u\n",y);
   printf("New value in z=%u\n",z);
   getch();
   }
```

Output:

```
 Value of i=3
Value of j=1.50
Value of k=c
Original value in x=65492
Original value in y=65494
Original value in z=65499
```

```
New value in x=65494
New value in y=65498
New value in z=65500
```

2.Assignment operator

consider the following example

int a=3,b=4;

int *p1=&a,*p2=&b,*p3=&a;

Then to check whether p1 and p3 are pointing at same address location ,equal to operator can be used.

if(p1==p3)

printf(“ p1 and p3 are pointing to same location”);

else

printf(“p1 and p3 are pointing to different location”);

Similarly to check whether p1 and p2 are pointing at different location, not equal to operator can be used.

if(p1!=p2)

printf(“ p1 and p2 are pointing to different location”);

else

printf(“p1 and p2 are pointing to same location”);

3. Assignment operator:

Consider the following

int a=3,*p1=&a,*p2;

p2=p1;

After the execution of the above , p2 and p1 will be pointing to same location.

Pointers to Pointers

In the c programming language, we have pointers to store the address of variables of any datatype. A pointer variable can store the address of a normal variable. C programming language also provides a pointer variable to store the address of another pointer variable. This type of pointer variable is called a pointer to pointer variable. Sometimes we also call it a double pointer. We use the following syntax for creating pointer to pointer…

datatype **pointerName ;

Example Program

int **ptr ;

Here, **ptr** is an integer pointer variable that stores the address of another integer pointer variable but does not stores the normal integer variable address.

MOST IMPORTANT POINTS TO BE REMEMBERED

1. To store the address of normal variable we use single pointer variable
2. To store the address of single pointer variable we use double pointer variable
3. To store the address of double pointer variable we use triple pointer variable
4. Similarly the same for remaining pointer variables also…

Example Program

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

int main()
{
  int a ;
  int *ptr1 ;
  int **ptr2 ;
  int ***ptr3 ;
```

```
  ptr1 = &a ;
  ptr2 = &ptr1 ;
  ptr3 = &ptr2 ;

  printf("\nAddress of normal variable 'a' = %u\n", ptr1) ;
  printf("Address of pointer variable '*ptr1' = %u\n", ptr2) ;
  printf("Address of pointer-to-pointer '**ptr2' = %u\n", ptr3) ;
   return 0;
}
```

Pointers to void in C

In the c programming language, pointer to void is the concept of defining a pointer variable that is independent of data type. In C programming language, a void pointer is a pointer variable used to store the address of a variable of any datatype. That means single void pointer can be used to store the address of integer variable, float variable, character variable, double variable or any structure variable. We use the keyword **"void"** to create void pointer. We use the following syntax for creating a pointer to void…

void *pointerName ;

Example Code

```
void *ptr ;
```

Here, **"ptr"** is a void pointer variable which is used to store the address of any datatype variable.

MOST IMPORTANT POINTS TO BE REMEMBERED

1. void pointer stores the address of any datatype variable.

Example Program

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

int main()
{
  int a =1;
  float b =3.5;
  char c =’a’;

  void *ptr ;

  ptr = &a ;
  printf(“Address of integer variable ‘a’ = %u\n”, ptr) ;
```

```
  printf("value of integer variable a is %d",*(int *)ptr);
  ptr = &b ;
  printf("Address of float variable 'b' = %u\n", ptr) ;
  printf("value of float variable a is %f",*(float *)ptr);

  ptr = &c ;
  printf("Address of character variable 'c' = %u\n", ptr) ;
  printf("value of character variable a is %c",*(char *)ptr);
}
```