ACHARYA INSTITUTE OF TECHNOLOGY

Soladevanahalli, Bengaluru – 560107

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Principles of Programming using C

(BPOPS203)

Prepared by

Mrs. Reshma

Assistant Professor,

Department of CS&E,

Acharya Institute of Technology

2024-2025

COs, POs, PEOs, PSOs

COURSE OUTCOMES (COS)

1. Illustrate the basic concepts of computer system and C programming fundamentals.
2. Apply the C programming constructs to solve simple problems.
3. Write C programs using arrays and functions to develop applications.
4. Apply the concepts of strings and pointer in writing the program for data manipulation and memory management.
5. Write C programs using structures, unions and Enumerated data-types to solve given problems.
6. Implement programs using C constructs, structures, strings, pointers and file I/O operations for given problems.

PROGRAM OUTCOMES (POS)

1. Engineering knowledge: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems

2. Problem analysis: Identify, formulate, review research literature, and analyse complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences

3. Design / development of solutions: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations

4. Conduct investigations of complex problems: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions

5. Modern tool usage: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modelling to complex engineering activities with an understanding of the limitations

6. The engineer and society: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

7. Environment and sustainability: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development

8. Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice

9. Individual and team work: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings

10. Communication: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions

11. Project management and Finance: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

12. Life-long learning: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change

PROGRAM EDUCATIONAL OBJECTIVES (PEOS)

PEO-1 Students shall, have a successful career in academia, R&D organizations, IT industry or pursue higher studies in specialized field of Computer Science and Engineering and allied disciplines.

PEO-2 Students shall, be competent, creative and a valued professional in the chosen field.

PEO-3 Students shall, engage in life-long learning, professional development and adapt to the working environment quickly

PEO-4 Students shall, become effective collaborators and exhibit high level of professionalism by leading or participating in addressing technical, business, environmental and societal challenges.

PROGRAM SPECIFIC OUTCOME (PSOS)

PSO-1 Students shall, apply the knowledge of hardware, system software, algorithms, networking and data bases.

PSO-2 Students shall, design, analyze and develop efficient, Secure algorithms using appropriate data structures, databases for processing of data.

PSO-3 Students shall, be Capable of developing stand alone, embedded and web-based solutions having easy to operate interface using Software Engineering practices and contemporary computer programming languages.

CO-PO -PSO Mapping

Outcomes	Program Outcomes													Program specific outcomes (PSOs)		
	1	2	3	4	5	6	7	8	9	10	11	12	RBT	1	2	3
BPOPS103.1	3				2				2			2	2			
BPOPS103.2	3	2	2		2				2	2		2	3			2
BPOPS103.3	3	2	2		2				2			2	3			2
BPOPS103.4	3	2	2		2				2	2		2	3			2
BPOPS103.5	3	2	2		2				2			2	3			2
BPOPS103.6	3	3	3		3				2	3		3	3			2

TABLE OF CONTENTS

Operators in C:

An operator is a symbol that specifies the mathematical, logical or relational operation to be performed. C language supports a lot of operators to be used in expressions. These operators can be categorized into the following major groups:

- Arithmetic operators
- Relational Operators
- Equality Operators
- Logical Operators
- Unary Operators
- Conditional Operators
- Bitwise Operators
- Assignment operators
- Comma Operator
- sizeof Operator

Arithmetic Operators:

Consider three variables declared as,
int a=9, b=3, result;
Below figure shows the arithmetic operators, their syntax, and usage in C language.

OPERATION	OPERATOR	SYNTAX	COMMENT	RESULT
Multiply	*	a * b	result = a * b	27
Divide	/	a / b	result = a / b	3
Addition	+	a + b	result = a + b	12
Subtraction	-	a - b	result = a - b	6
Modulus	%	a % b	result = a % b	0

Arithmetic operators can be applied to any integer of floating-point numbers. The operator % (modulus) finds the remainder of an integer division. This operator can be applied only to integer operands and cannot be used on float or double operands.

When both operands of the division operator (/) are integers, the division is performed as an integer division. Integer division always results in an integer result. So, the result is always rounded-off by ignoring the remainder.

Therefore,

$$9/4 = 2 \text{ and } -9/4 = -3$$

If op1 and op2 are integers ad the quotient is not an integer, then we have two cases:

1. If op1 and op2 have the same sign, then op1/op2 is the largest integer less thetrue quotient.
2. If op1 and op2 have opposite signs, then op1/op2 is the smallest integer greaterthan the true quotient.

Note that it is not possible to divide any number by zero. This is an illegal operation that results in a run-time division-by-zero exception, thereby terminating the program.

Relational Operator:

Relational operator is also known as a comparison operator, it is an operator that compares two values. Expressions that contain relational operators are called *relational expressions*. Relational operators return true or false value, depending on whether the conditional relationship between the two operands holds or not.

For example, to test if x is less than y, relational operator < is used as x < y. This expression will return true (1) if x is less than y; otherwise the value of the expression will be false (0).

OPERATOR	MEANING	EXAMPLE
<	LESS THAN	3 < 5 GIVES 1
>	GREATER THAN	7 > 9 GIVES 0
>=	LESS THAN OR EQUAL TO	100 >= 100 GIVES 1
<=	GREATER THAN EQUAL TO	50 >=100 GIVES 0

When arithmetic expressions are used on either side of a relational operator, then first the arithmetic expression will be evaluated and then result will be compared. This is because arithmetic operators have higher priority over relational operators.

Equality Operators:

C language supports two kinds of equality operators to compare their operands forstrict equality or inequality. They are equal to (==) and not equal to (!=) operator.

The equality operators have lower precedence than the relational operators.

- The equal-to operator (==) returns true (1) if operands on both sides of theoperator have same value; otherwise, it returns false (0).
- On the contrary, the not-equal-operator (!=) returns true (1) if the operands donot have the same values; else it returns false (0).

OPERATOR	MEANING
==	RETURNS 1 IF BOTH OPERANDS ARE EQUAL, 0 OTHERWISE
!=	RETURNS 1 IF OPERANDS DO NOT HAVE THE SAME VALUE, 0 OTHERWISE

Logical Operators:

- C language supports three logical operators. They are- Logical AND (&&), LogicalOR (||) and Logical NOT (!).
- As in case of arithmetic expressions, the logical expressions are evaluated fromleft to right.

Logical AND

Logical AND operator is a binary operator, which simultaneously evaluates two values or relational expressions. If both the operands are true, then the whole expression evaluates to true. If both or one of the operands is false, then the whole expression evaluates to false.

A	B	A &&B
0	0	0
0	1	0
1	0	0
1	1	1

Logical OR

Logical OR returns a false value of both the operands are false. Otherwise it returns atrue value.

A	B	A || B
0	0	0
0	1	1
1	0	1
1	1	1

Logical NOT

The logical NOT operator takes a single expression and negates the value of the expression. That is, logical NOT produces a 0 if the expression evaluates to a non-zero value and produces a 1 if the expression produces a zero. In other words, it justreverses the value of the expression.

A	! A
0	**1**
1	**0**

Unary Operators:

- Unary operators act on **single operands**. C language supports **three unary operators**. They are **unary minus, increment and decrement operators**.
- When an operand is **preceded by a minus sign, the unary operator negates its value**.
- The **increment operator is a unary operator that increases the value of its operand by 1**.
- Similarly, the decrement operator decreases the **value of its operand by 1**.

For example,
int x = 10, y;
y = x++;
is equivalent to writing
y = x;
x = x + 1; whereas, y = ++x;
is equivalent to writing
x = x + 1;
y = x;

Conditional Operators:

- The conditional operator, **operator (?:) is just** like an **if .. else statement that can be written within expressions**.
- The syntax of the conditional operator is **exp1 ? exp2 : exp3**
- Here, **exp1 is evaluated first**. If it is **true then exp2 is evaluated** and becomes the result of the expression, otherwise **exp3 is evaluated and becomes the result of the expression**.
  For example,
  large = (a > b) ? a : b
- Conditional operators make the **program code more compact, more readable, and safer to use as it is easier both to check and guarantee** that the arguments that are used for evaluation.
- **Conditional operator** is also known as **ternary operator as it is neither a unary nor a binary operator**; it takes ***three*** **operands**.

Bitwise Operators:

Bitwise operators perform operations at bit level. These operators include:
bitwise AND, bitwise OR, bitwise XOR and shift operators.

- The bitwise AND operator (&) is a small version of the boolean AND (&&) as it performs operation on bits instead of bytes, chars, integers, etc.
- The bitwise OR operator (|) is a small version of the boolean OR (||) as it performs operation on bits instead of bytes, chars, integers, etc.
- The bitwise NOT (~), or complement, is a unary operation that performs logical

negation on each bit of the operand. By performing negation of each bit, it actually produces the ones' complement of the given binary value.

- The bitwise XOR operator (^) performs operation on individual bits of the operands. The result of XOR operation is shown in the table.

Bitwise Shift Operators:

In bitwise shift operations, the digits are moved, or *shifted*, to the left or right.

The CPU registers have a fixed number of available bits for storing numerals, so when we perform shift operations; some bits will be "shifted out" of the register at one end, while the same number of bits are "shifted in" from the other end.

In a left arithmetic shift, zeros are shifted in on the right. For example;

unsigned int x = 11000101; Then

x << 2 = 00010100

If a right arithmetic shift is performed on an unsigned integer then zeros are shifted on the left.

unsigned int x = 11000101;

Then

x >> 2 = 00110001

Assignment Operators:

The assignment operator is responsible for assigning values to the variables.

While the equal sign (=) is the fundamental assignment operator, C also supports other assignment operators that provide shorthand ways to represent common variable assignments. They are shown in the table.

The assignment operator has right-to-left associativity,

So the expression

a = b = c = 10; is evaluated as (a = (b = (c = 10)))

OPERATOR	SYNTAX	EQUIVALENT TO
/=	variable /= expression	variable = variable / expression
*=	variable *= expression	variable = variable * expression
+=	variable += expression	variable = variable + expression
-=	variable -= expression	variable = variable - expression
&=	variable &= expression	variable = variable & expression
^=	variable ^= expression	variable = variable ^ expression
<<=	variable <<= amount	variable = variable << amount
>>=	variable >>= amount	variable = variable >> amount

Comma Operator:

The comma operator in **C takes two operands**. It works by evaluating the **first and discarding its value**, and then evaluates the **second and returns** the value as the result of the expression.

Comma separated operands when chained together are evaluated in **left-to-right sequence** with the right-most value yielding the result of the expression.

Among all the operators, the **comma operator** has the lowest precedence.

For example,

int a=2, b=3, x=0; x = (++a, b+=a);

Now, the value of x = 6.

sizeof Operator:

sizeof is a unary operator used to calculate the sizes of data types. It can be applied to **all data types**.

The operator returns the size of the variable, data type or expression in bytes.

'**sizeof**' operator is used to determine the amount of memory space that the **variable/expression/data type will take**.

For example,

sizeof(char) returns 1, that is the size of a character data type.

If we have,

int a = 10;

unsigned int result;

result = sizeof(a); then result = 2

C operators have two properties: priority and associativity. When the expression has more than one operator then it is the relative priority of the operator with respect to each other that determine the order in which the expression will be evaluated. Associativity defines the direction in which the operator acts on the operands. It can be either left-to-right or right-to-left. Priority is given precedence over associativity to determine the order in which the expressions are evaluated. Associativity is then applied, if the need arises.

Rules for Evaluation of Expression

1. First, parenthesized sub expressions from left to right are evaluated.
2. If parentheses are nested, the evaluation begins with the innermost sub- expression.
3. The precedence rule is applied in determining the order of application of operators in evaluating sub-expressions.
4. The associativity rule is applied when two or more operators of the same precedence level appear in a sub-expression.
5. Arithmetic expressions are evaluated from left to right using the rules of precedence.
6. When parentheses are used, the expressions within parentheses assume highest priority.

Operator	Associativity	Operator	Associativity
() [] . —>	left-to-right	&	left-to-right
++(postfix) – –(postfix)	right-to-left	^	left-to-right
++(prefix) – –(prefix) +(unary) – (unary) ! ~ (type) *(indirection) &(address) sizeof	right-to-left	|	left-to-right
* / %	left-to-right	&&	left-to-right
+ –	left-to-right	||	left-to-right
<< >>	left-to-right	?:	right-to-left
< <= > >=	left-to-right	= += –= *= /= %= &= ^= |= <<= >>=	right-to-left
== !=	left-to-right	,(comma)	left-to-right

Evaluating expressions using the precedence chart

1. x = 3 * 4 + 5 * 6
   = 12 + 5 * 6
   = 12 + 30
   = 42

```
2. x = 3 * (4 + 5) * 6
     = 3 * 9 * 6
     = 27 * 6
     = 162
3. x = 3 * 4 % 5 / 2
     = 12 % 5 / 2
     = 2 / 2
     = 1
4. x = 3 * (4 % 5) / 2
     = 3 * 4 / 2
     = 12 / 2
     = 6
5. x = 3 * 4 % (5 / 2)
     = 3 * 4 % 2
     = 12 % 2
     = 0
6. x = 3 * ((4 % 5) / 2)
     = 3 * (4 / 2)
     = 3 * 2
     = 6
```

Take the following variable declarations,

```
int a = 0, b = 1, c = -1;
float x = 2.5, y = 0.0;
```

If we write,

```
a = b = c = 7;
```

Since the assignment operator works from right-to-left, `c = 7`. Then since `b = c`, therefore `b = 7`. Now `a = b`, so `a = 7`.

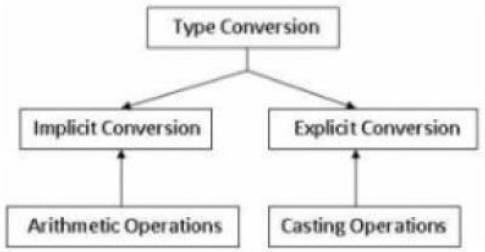
TYPE CONVERSION

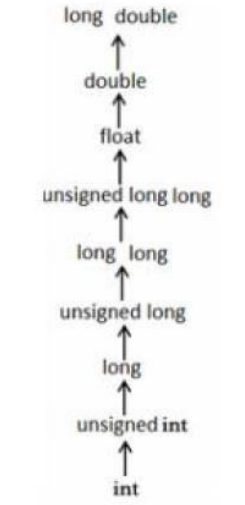
- Type conversion is used to convert data of one type to data of another type.
- Type conversion is of 2 types as shown in below figure:

IMPLICIT TYPE CONVERSION

C permits mixing of constants and variables of different types in an expression. C automatically converts any intermediate values to the proper type so that the expression can be evaluated without losing any significance. This automatic conversion is known as *implicit type Conversion*.

During evaluation it adheres to very strict rules of type conversion. If the operands are of different types, the „lower" type is automatically converted to the higher type before the operation proceeds. The result is of the higher type.





datatype hierarchy

The sequence of rules that are applied while evaluating expressions. All short and char are automatically converted to int; then

1. If one of the operands is **long double**, the other will be converted to **long double** and the result will be **long double**;
2. Else, if one of the operands is **double**, the other will be converted to **double** and the result will be double;
3. Else, if one of the operands is **float**, the other will be converted to **float** and result will be **float**;
4. Else, if one of the operands is **unsigned long int**, the other will be converted to **unsigned long int** and result will be **unsigned long int**;
5. Else, if one of the operands is **long int**, the other will be converted to **unsigned int,** then
    - If **unsigned int** can be converted to **long int**, the **unsigned int** operand will be converted as such and the result will be **long int**;
    - Else, both operands will be converted to **unsigned long int** operand will be **unsigned long int**;
6. Else, if one of the operands is **long int**, the other will be converted to **long int** and result will be **long int**;
7. Else, if one of the operands is **unsigned int**, the other will be converted to **unsigned int** and result will be **unsigned int**;

Note that some versions of C automatically convert all floating-point operands to double precision

The final result of an expression is converted to the type of the variable on the left of the assignment sign before assigning the value to it. However, the following changes are introduced during the final assignment.

1. **float** to **int** causes truncation of the fractional part.
2. **double** to **float** causes rounding of digits.
3. **long int** to **int** causes dropping of the excess higher order bits.

```
//Example: Program to illustrate implicit conversion.

#include<stdio.h>
void main()
{
    int a = 22, b=11; float d ;
    d=b/c;
    printf("d Value is : %f ", d );
}

Output:
d Value is : 0.500000
```

EXPLICIT CONVERSION

- When the data of one type is converted explicitly to another type with the help of some pre-defined functions, it is called as explicit conversion.
- There may be data loss in this process because the conversion is forceful.
- The syntax is shown below:

data_type1 v1;

data_type2 v2= (data_type2) v1;

where v1 can be expression or variable

- For ex:

float b=11.000000; int c = 22;

float d = b/(float)c =11.000000/22.000000 = 0.500000

```
//Example: Program to illustrate explicit conversion.

#include<stdio.h>
void main()
{
    float b=11.000000;
    int c = 22;
    float d; d=b/(float)c;
    printf("d Value is : %f ", d );
}


Output:
d Value is : 0.500000
```

Feature	Type Conversion	Type Casting
Also Known As	Implicit Type Conversion	Explicit Type Conversion
How it happens	Automatically by the compiler	Manually by the programmer
Syntax	No special syntax	Uses (type) syntax
Risk of data loss	Rare, unless converting to smaller type	Yes, possible — programmer takes responsibility
Example	int + float → float	(float)intVar

Decision control and Looping statements

Introduction to Decision Control Statements

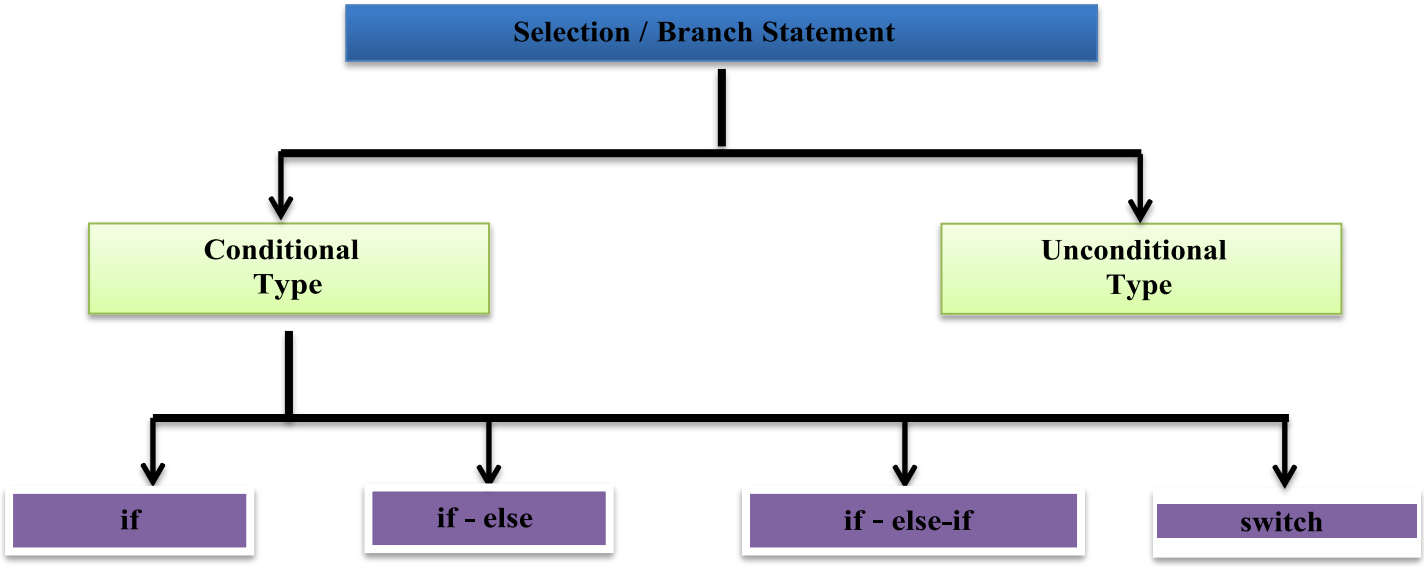
C program is executed sequentially from the first line of the program to its last line, i.e., the second statement is executed after the first, the third statement is executed after the second, and so on.

Although this is true, but in some cases we want only selected statements to be executed. Such type of conditional processing extends the usefulness of programs.

C supports two type of decision control statements that can alter the flow of a sequence of instructions. These include conditional type branching and unconditional type branching. Below figure shows the categorization of decision control statements in C language.

Conditional Branching Statements

- Decision control statements are used to **alter the flow of a sequence** of instructions.
- These statements help to **jump from one part of the program to another** depending on whether a **particular condition is satisfied or not.**
- These decision control statements include:
  - **Simple If statement**
  - **If else statement**
  - **If else if statement**
  - **Switch statement**

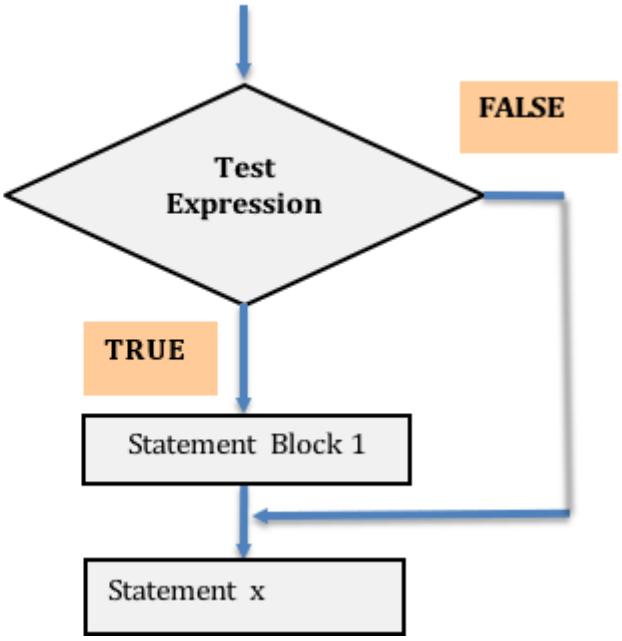


Simple if statement

- If statement is the **simplest form of decision control** statements that is frequently used in **decision making**. The general form of a simple if statement is shown in the figure.
- First the test **expression is evaluated**. If the test expression is **true**, the statement of if block (**statement 1 to n**) are executed otherwise these **statements will be skipped** and the execution will **jump to statement x**.

SYNTAX OF IF STATEMENT

```
if (test expression)
{
        statement 1;
        ..............
        statement n;
}
statement x;
```




```
/* Program to Check the given number is Negative or Positive */
#include<stdio.h>
void main()
{
    int n;
    printf("Enter any non zero integer: \n") ;
    scanf("%d", &n);
    if(n>0)
        printf("Number is positive number");

    if(n<0)
        printf("Number is negative number");
}
```

if-else Statement:

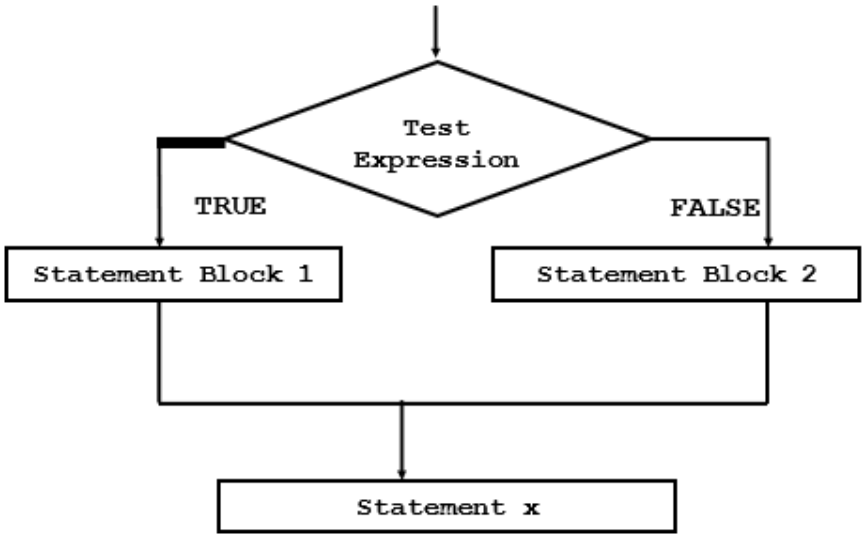
This is basically a "two-way" decision statement.

This is used when we must choose between two alternatives.

In the if-else construct, first the **test expression is evaluated**. If the expression is true, statement block 1 is executed and statement block 2 is skipped. Otherwise, if the expression is false, statement block 2 is executed and statement block 1 is ignored. In any case after the statement block 1 or 2 gets executed the control will pass to statement x. Therefore, statement x is executed in every case.

SYNTAX OF IF STATEMENT

```
if (test expression)
{
     statement block 1;
}
else
{
     statement block 2;
}

statement x;
```

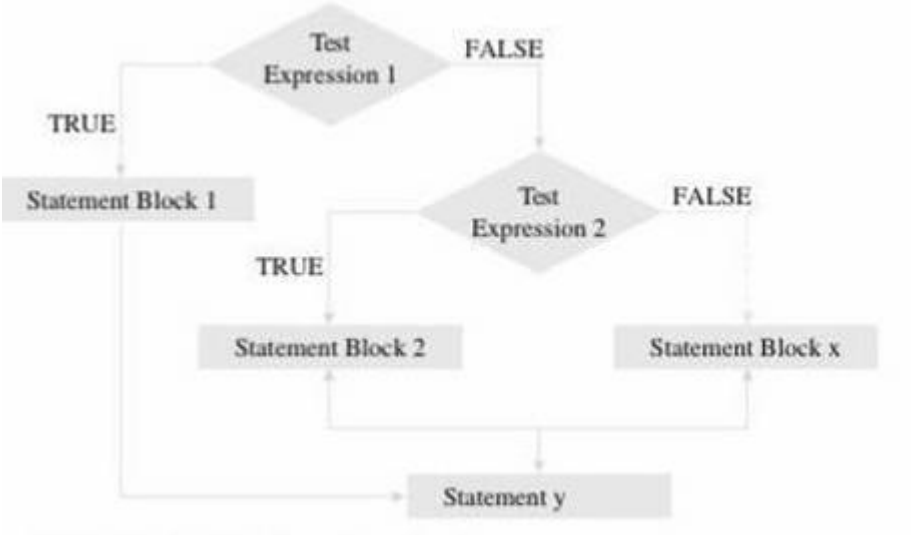


Example Program:

```
// PROGRAM TO FIND WHETHER A NUMBER IS EVEN OR ODD
#include<stdio.h>
void main()
{
    int a;
    printf("\n Enter the value of a : ");
    scanf("%d", &a);

    if(a%2==0)
        printf("\n %d is even", a);

    else
        printf("\n %d is odd", a);

}
```

if-else-if Statement (if-else-if Ladder in C):

- C language supports if else if statements to test additional conditions apart from the initial test expression. The if-else-if construct works in the same way as a normal if statement.
- if-else-if construct is also known as nested if construct. It is illustrated in the below figure.



SYNTAX OF IF-ELSE STATEMENT

```
if ( test expression 1)
{
        statement block 1;
}
else if ( test expression 2)
{
        statement block 2;
}
.....
else if (test expression N)
{
        statement block N;
}
else
{
        Statement Block X;
}
Statement Y;
```

Example programs:

```
#include <stdio.h>

int main() {
   int i = 20;

   // If else ladder with three conditions
   if (i == 10)
      printf("Not Eligible");
   else if (i == 15)
      printf("wait for three years");
   else if (i == 20)
      printf("You can vote");
   else
      printf("Not a valid age");

   return 0;
}
```

```c
//Electricity bill generation program- refer lab program 3 in the VTU syllabus

#include <stdio.h>
int main() {
    char name[50];
    int units;
    float charge = 0, total_charge, surcharge = 0;

    // Input
    printf("Enter user name: ");
    scanf("%s",&name);

    printf("Enter number of units consumed: ");
    scanf("%d", &units);

    // Calculate basic charge based on unit slabs
    if (units <= 200)
    {
        charge = units * 0.80;
    }
    else if (units <= 300)
    {
        charge = 200 * 0.80 + (units - 200) * 0.90;
    }
    else
    {
        charge = 200 * 0.80 + 100 * 0.90 + (units - 300) * 1.00;
    }

    // Add minimum meter charge
    total_charge = charge + 100;


    // Apply surcharge if applicable
    if (total_charge > 400)
    {
        surcharge = total_charge * 0.15;
        total_charge += surcharge;
    }

    // Output
    printf("\n--- Electricity Bill ---\n");
    printf("Name: %s\n", name);
    printf("Units Consumed: %d\n", units);
    printf("Surcharge: Rs. %.2f\n", surcharge);
    printf("Total Amount to Pay: Rs. %.2f\n", total_charge);

    return 0;
}
```

Rules for Indentation:

When using control structures, a statement often controls many other statements that follow it. In such situations it is a good practice to use indentation to show that the indented statements are dependent on the preceding controlling statement. Some guidelines that could be followed while using indentation are listed below:

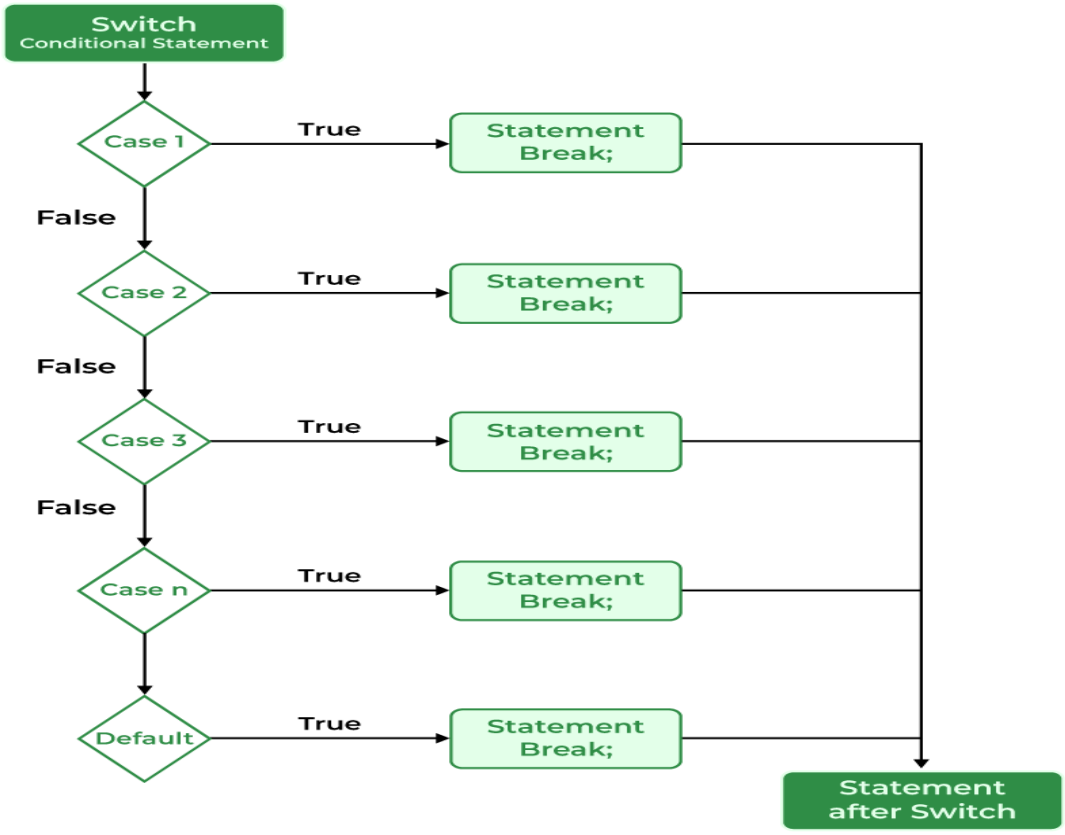
- Indent statements that are dependent on the previous statements; provide atleast three spaces of indentation.
- Align vertically else clause with their matching of clause.
- Use braces on separate lines identify a block of statements.
- Indent the statements in the block by at least three spaces to the right of thebraces.
- Align the opening and closing braces.
- Use appropriate comments to signify the beginning and end of the block.
- Indent the nested statements as per the above rules.
- Code only one clause or statement on each line.

switch case:

This is an alternative to the if else if ladder that can be used to execute the conditional code based on the value of the variable specified in the switch statement. The switch block consists of cases to be executed based on the value of the switch variable.

Syntax of switch

```
switch (expression) {
  case value1:
    // Code_block1
    break;
  case value2:
    // Code_block2
    break;
  default:
    // Default_code_block
}
```



Example:

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

  // variable to be used in switch statement
   int var = 18;

   // declaring switch cases
   switch (var) {
   case 15:
      printf("You are a kid");
      break;
   case 18:
      printf("Eligible for vote");
      break;
   default:
      printf("Default Case is executed");
      break;
   }
   return 0;
}
```

Advantages of Using a switch case statement:
Switch case statement is performed by programmers due to the following reasons:
1.Easy to debug
2.Easy to read and understand
3.Ease of maintenance as compared with its equivalent if-else statement
4.Like if-else statements, switch statements can also be nested.
5.Executes faster than its equivalent if-else statement.

ITERATIVE STATEMENTS

Iterative statements are used to repeat the execution of a list of statements, depending on the value of an integer expression. C language supports three types of iterative statements also known as looping statements. They are:

- While loop
- Do-while loop
- For loop

while Loop

- The while loop is used to repeat one or more statements while a particular condition is true.
- Below figure shows the syntax and general form representation of a while loop.
- In the while loop, the condition is tested before any of the statements in the statement block is executed.
- If the condition is true, only then the statements will be executed otherwise the control will jump to the immediate statement outside the while loop block.
- We must constantly update the condition of the while loop.

Syntax of While Loop

```
statement  x;
while  (condition)
{
 statement block;
}
statement  y;
```

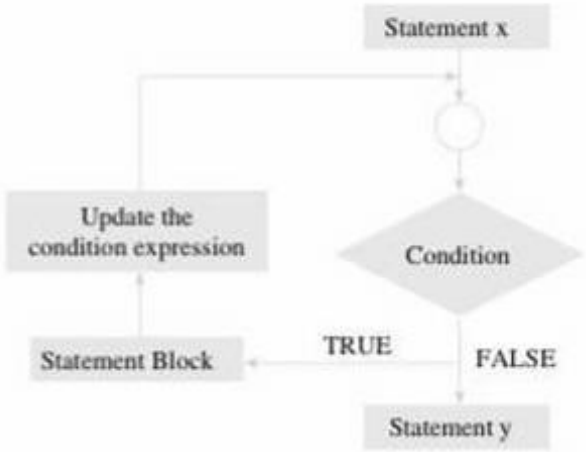


Figure 10.6 The while loop construct

```
//program to find sum of 1st 10 natural numbers

#include <stdio.h>
int main()
{
    int i=1, sum=0;
    while(i<=n)
    {
        sum=sum+i;
        i=i+1; //update expression
    }
    printf("SUM = %d",sum);
    return 0;

}
```

Do-while Loop

- The do-while loop is similar to the while loop. The only difference is that in a do- while loop, the test condition is tested at the end of the loop.
- Note that the test condition is evaluated at the end, this clearly means the body of the loop gets executed at least one time (even if the condition is false).
- The do while loop continues to execute whilst a condition is true. There is no choice whether to execute the loop or not. Hence, entry in the loop is automatic there is only a choice to continue it further or not.
- The major disadvantage of using a do while loop is that it always executes at least once, so even if the user enters some invalid data, the loop will execute.

- Do-while loops are widely used to print a list of options for a menu driven program.

Syntax of do-while Loop

```
statement x;
do
{
statement block;
}while (condition);

statement y;
```

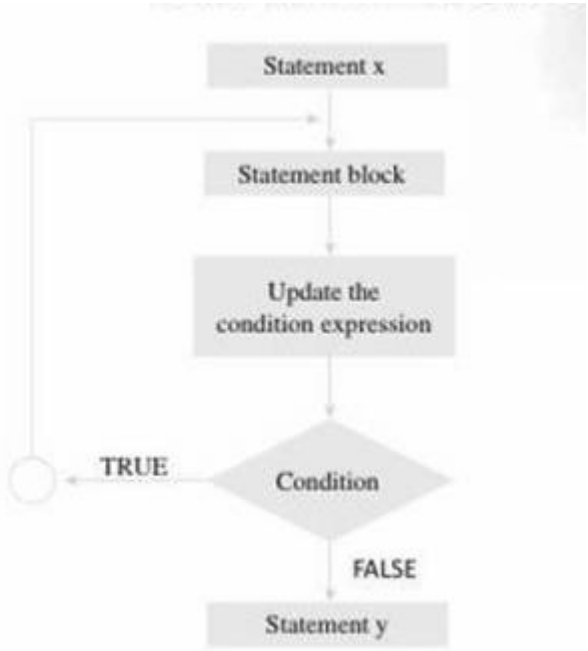


Figure 10.7 do-while construct

```
// PROGRAM TO PRINT NUMBERS FROM 0-10 USING DO-WHILE LOOP

#include<stdio.h>
int main()
{
    int i = 0;
    do
    {
        printf("\n %d", i); i = i + 1;
    }
    while(i<=10);

}
```

for Loop

Like the while and do-while loops, the for loop provides a mechanism to repeat a task until a particular condition is true. For loop is usually known as a deterministic or definite loop because the programmer knows as a exactly how many times the loop will repeat. The general syntax and form of for loop is shown below.

Syntax of for Loop

```
for (initialization;  condition;
     increment/decrement/update)
{
statement block;
}
statement y;
```

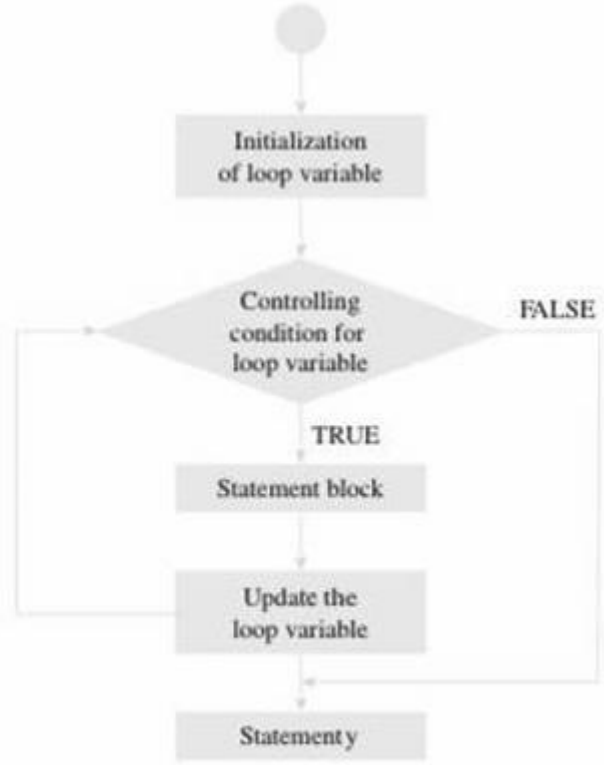


Figure 10.8 for loop construct

- When a for loop is used, the loop variable is initialized only once.
- With every iteration of the loop, the value of the loop variable is updated and the condition is checked. If the condition is true, the statement block of the loop isexecuted else, the statements comprising the statement block of the for loop are skipped and the control jumps to the immediate statement following the for loop body.
- Updating the loop variable may include incrementing the loop variable, decrementing the loop variable or setting it to some other value like, i +=2, wherei is the loop variable.

Comparison of the Three Loops

No.	Topics	For loop	While loop	Do...while loop
01	Initialization of condition variable	In the parenthesis of the loop.	Before the loop.	Before the loop or in the body of the loop.
02	Test condition	Before the body of the loop.	Before the body of the loop.	After the body of the loop.
03	Updating the condition variable	After the first execution.	After the first execution.	After the first execution.
04	Type	Entry controlled loop.	Entry controlled loop.	Exit controlled loop.
05	Loop variable	Counter.	Counter.	Sentinel & counter

For loop	While loop	Do.....while loop

for(n = 1; n <= 10; n++)
{
========
========
}
	n = 1;
	while(n <=10)
	{
	==========
	==========
	n=n+1;
	}
		n = 1;
		do
		{
		========
		========
		n = n + 1;
		}
		while(n<=10);

Nested Loops

C allows its users to have nested loops, i.e., loops that can be placed inside other loops. Although this feature will work with any loop such as while, do-while, and for, it is most commonly used with the for loop, because this is easiest to control.

A for loop can be used to control the number of times that a particular set of statements will be executed. Another outer loop could be used to control the number of times that a whole loop is repeated.

Syntax

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

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

Example: Program to print triangles using *, numbers and characters

```
*
* *
* * *
* * * *
* * * * *
```

```c
#include <stdio.h>
void main()
{
    int i, j, rows;
    printf("Enter number of rows: ");
    scanf("%d",&rows);
        for(i=1; i<=rows; i++)
        {
            for(j=1; j<=i; j++)
            {
                printf("* ");
            }
        printf("\n");
        }
}
```

```
Enter number of rows: 5
*
* *
* * *
* * * *
* * * * *

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

Example 2: Program to print half pyramid a using numbers

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

main.c

```c
#include <stdio.h>
void main()
{
    int i, j, rows;
    printf("Enter number of rows: ");
    scanf("%d",&rows);
        for(i=1; i<=rows; i++)
        {
            for(j=1; j<=i; j++)
            {
                printf("%d",j);
            }
        printf("\n");
        }
}
```

```
Enter number of rows: 5
1
12
123
1234
12345


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

Programs to print inverted half pyramid using * and numbers

main.c

```c
#include <stdio.h>
void main()
{
    int i, j, rows;
    printf("Enter number of rows: ");
    scanf("%d",&rows);
        for(i=rows; i>=1; i--)
        {
            for(j=1; j<=i; j++)
            {
                printf("*");
            }
        printf("\n");
        }
}
```

```
Enter number of rows: 5
*****
****
***
**
*


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

main.c

```c
#include <stdio.h>
void main()
{
    int i, j, rows;
    printf("Enter number of rows: ");
    scanf("%d",&rows);
        for(i=rows; i>=1; i--)
        {
            for(j=1; j<=i; j++)
            {
                printf("%d",j);
            }
        printf("\n");
        }
}

```

```
Enter number of rows: 5
12345
1234
123
12
1


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

Break and continue Statement:

Loops perform a set of operations repeatedly until the control variable fails to satisfy the test-condition. The number of times a loop is repeated is decided in advance and the test condition is written to achieve this. Sometimes, when executing a loop it becomes desirable to skip a part of the loop or to leave the loop as soon as a certain condition occurs.

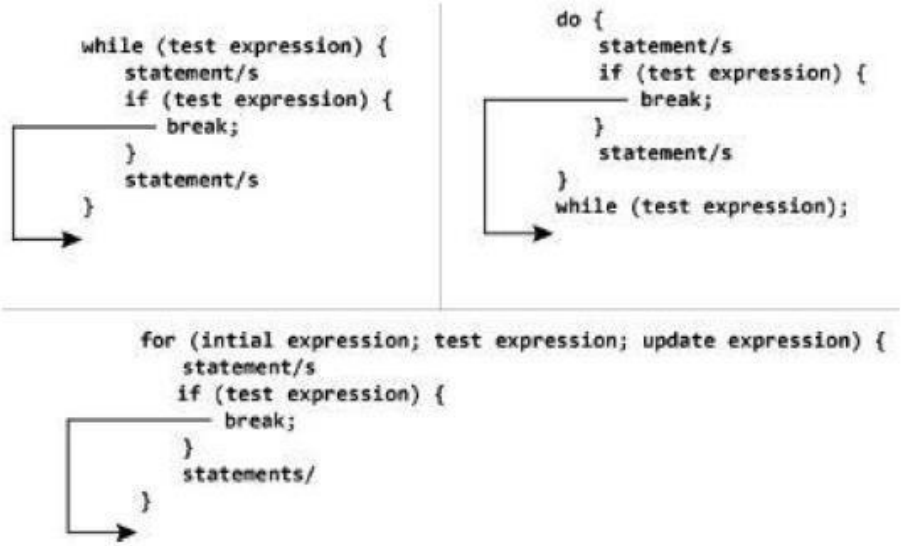
Consider a searching program, loop written for reading and testing the names 100 times must be terminated as soon as the desired name if found. C permits a jump from one statement to another statement within a loop as well as a jump out of a loop.

THE break STATEMENT

- The break statement is jump statement which can be used in switch statement andloops.
- The break statement works as shown below:

1) If break is executed in a switch block, the control comes out of the switch block and the statement following the switch block will be executed.

2) If break is executed in a loop, the control comes out of the loop and the statementfollowing the loop will be executed.



Example:

main.c

```c
#include <stdio.h>
void main()
{
    int i;
    for(i=1;i<=10;i++)
    {
        if(i==5)
        {
            break;
        }
        printf("%d",i);
    }

    printf("\nOutside for");
}
```

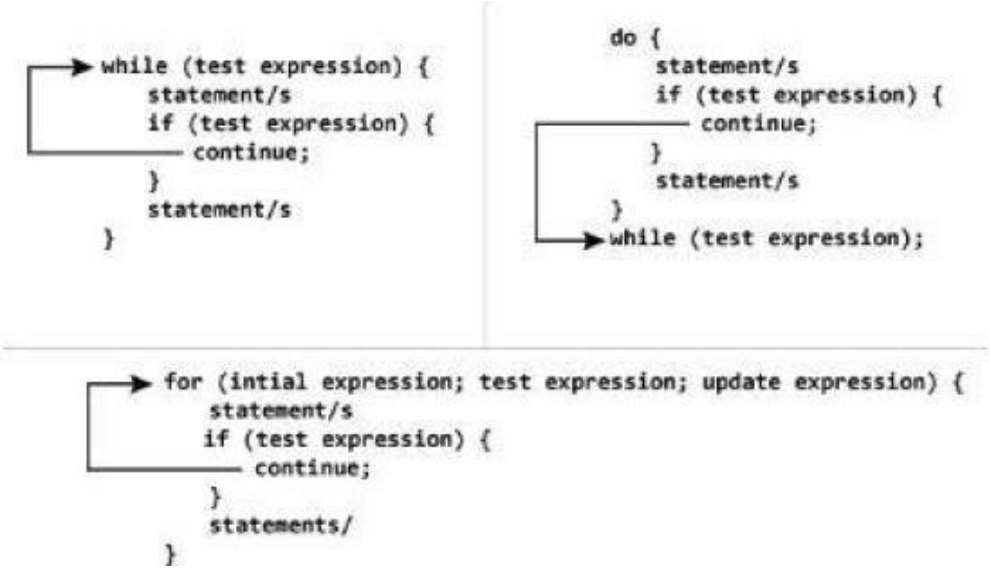
```
1234
Outside for

...Program finished with exit code 12
Press ENTER to exit console.
```

THE continue STATEMENT

During execution of a loop, it may be necessary to skip a part of the loop based onsome condition.

In such cases, we use continue statement.

- The continue statement is used only in the loop to terminate the current iteration.
- The syntax is shown below:



Example:

main.c

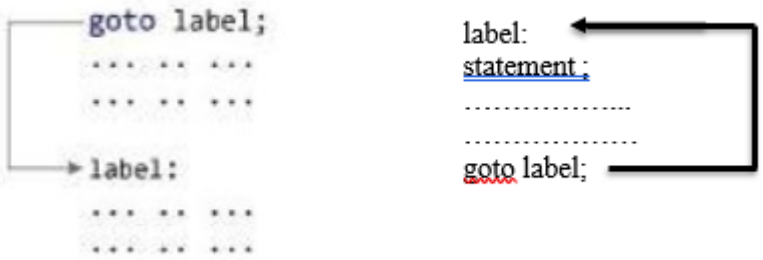
```c
#include <stdio.h>
void main()
{
    int i;
    for(i=1;i<=10;i++)
    {
        if(i==5)
        {
            continue;
        }
        printf("\n%d",i);
    }

    printf("\nOutside for");
}
```

```
1
2
3
4
6
7
8
9
10
Outside for

...Program finished with exit code 12
Press ENTER to exit console.
```

goto Statement:

- goto statement can be used to branch unconditionally from one point to another in the program.
- The goto requires a label in order to identify the place where the branch is to be made.
- A label is any valid variable name and must be followed by a colon(:).
- The label is placed immediately before the statement where the control is to be transferred.

- The syntax is shown below:



The label: can be anywhere in the program either before or after the goto label; statement. During running of a program when a statement like

goto begin;

Note that a goto breaks the normal sequential execution of the program. If the label: is before the statement goto label; a loop will be formed and some statements will be executed repeatedly. Such a jump is known as a ***backward jump***. On the other hand, if the label: is placed after the goto label; some statements will be skipped and jump is known as a ***forward jump.***

Example: Program to detect the entered number as to whether it is even or odd. Use goto statement.

```
#include<stdio.h>
void main()
{
    int x;
    printf("Enter a Number: \n");
    scanf("%d", &x);
    if(x % 2 = = 0)
        goto even;
    else
        goto odd;
    even: printf("%d is Even Number");
          return;
    odd: printf(" %d is Odd Number");
}
```