Decision Making and Branching

- C program is a set of statements which are normally executed sequentially in the order in which they appear
- Happens when no options or no repetitions of certain calculations.
- Number of situations where statement execution order is changed on the basis certain conditions, or repeat a group of statements until certain specified condition is satisfied.
- Decision making and branching statements are utilized to check whether a condition has occurred or not
- C language possesses such decision-making:

1. **if-else** statement
2. **switch** statement
3. **Conditional operator** statement
4. **goto** statement

Decision Making with *if* Statement

- **if** is a powerful decision-making statement and is used to control flow of execution statements.
- Basically a two-way decision statement and is used in conjunction with an expression.
- **Syntax:**

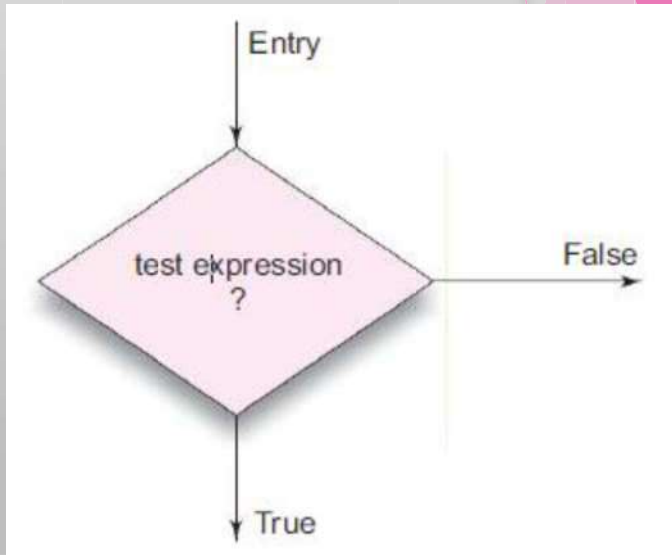
if (test expression)

- ***Allows compiler to evaluate expression first and then, depending on whether the value of the expression (relation or condition) is 'true' (or non-zero) or 'false' (zero), it transfers the control to particular statement***
- This point of program has two paths to follow, one for the true condition and the other for the false condition
- if statement may be implemented in different forms

depending on the complexity of conditions to be tested.

The different forms are:

1. Simple **if** statement
2. **if….else** statement
3. **Nested if…..else** statement
4. **else if** laddar



Simple if Statement

- General form of a simple ***if*** statement is

if (test expression)

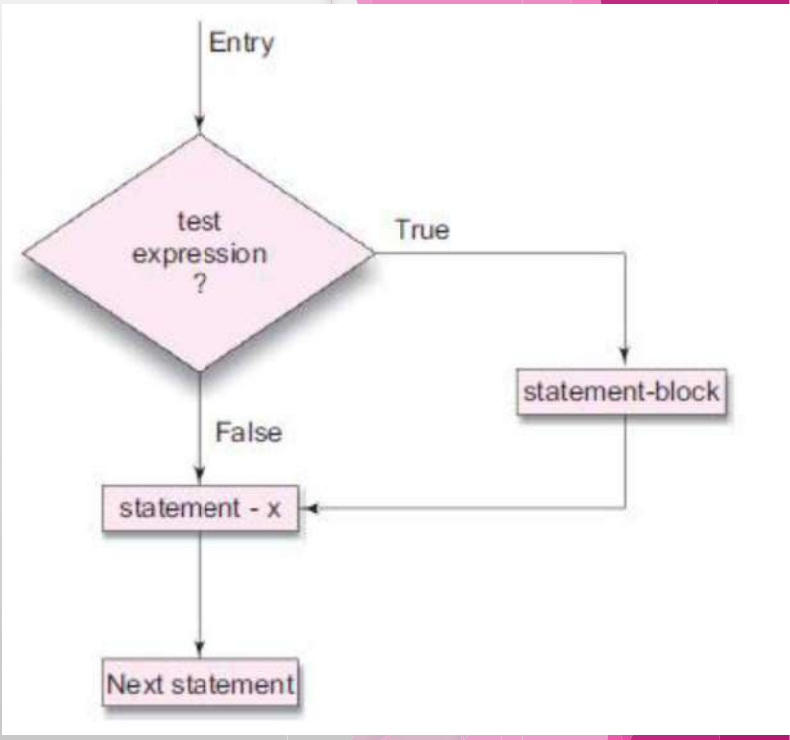
{

Statement-block;

}

Statement-X;

- ***'statement-block'*** may be a single statement or a group of statements
- **If test expression** is **true**, **statement-block** will be **executed;** otherwise the statement-block will be skipped and the **execution** will **jump to** the **statement-x**.
- When **condition is true both** the **statement-block** and **statement-x** are executed in sequence



Applying De Morgan's Rule

- In decision statements, often come across a scenario where the **logical NOT operator** is applied to a compound logical expression, like **!(x&&y||!z)**
- **Positive logic easy** to **read and comprehend** than a **negative logic.** We apply **De Morgan's Law** to make the total expression positive.
- Rule is as follows:
- **"Remove the parentheses by applying the NOT operator to every logical expression component, while complementing the relational operators"**

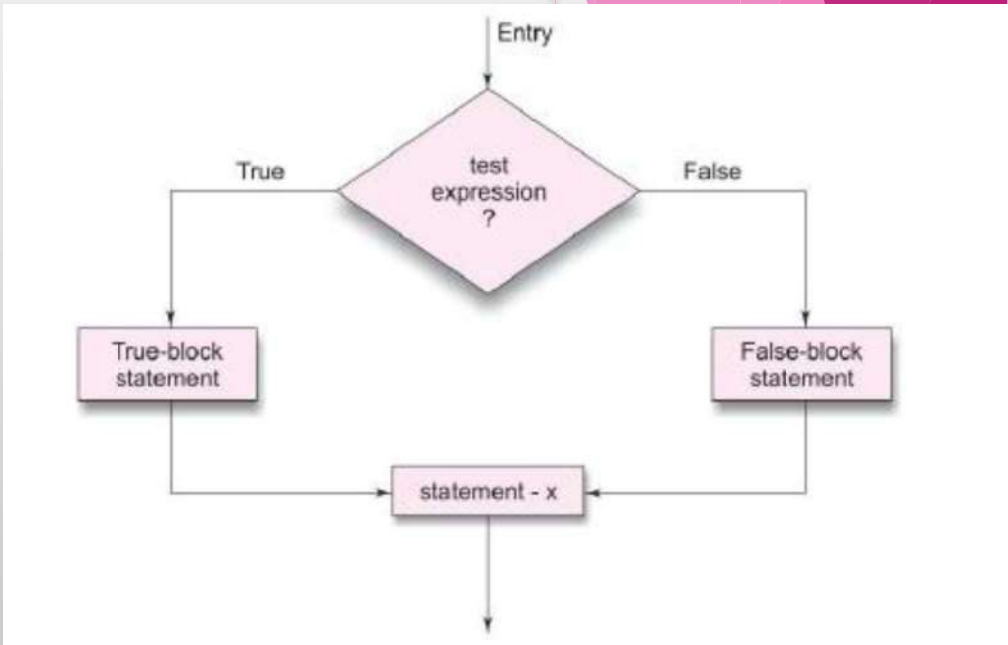
1. **x becomes !x**
2. **!x becomes x**
3. **&& becomes ||**
4. **|| becomes &&**

- **Examples:**
  - ***!(x && y || !z) becomes !x || !y && z,***
  - ***!(x < = 0 || !condition) becomes x >0&& condition***

if else Statement

- **if…else** statement is an extension of the simple **if** statement
- **General form:**

```
if (test expression)
{
True-block statements
}
else
{
False-block statement
}
Statement-x
```

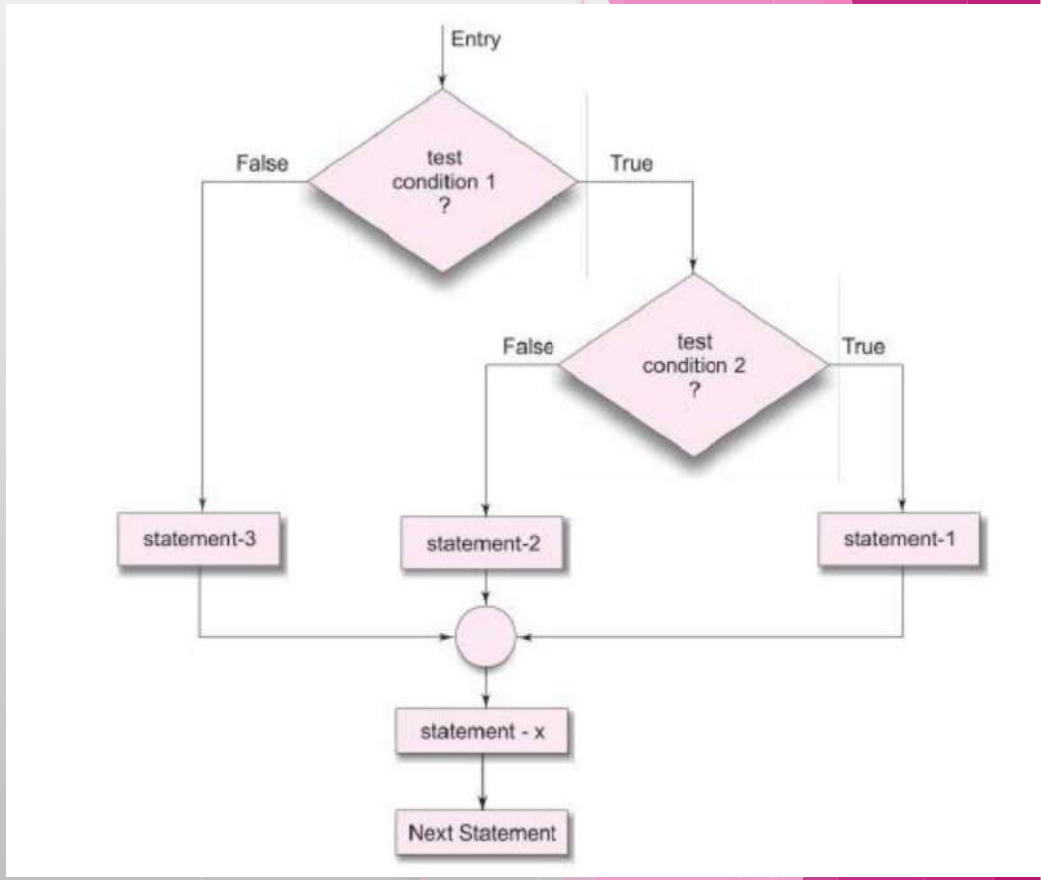


- If the test expression is true, then the true-block statement(s), immediately following the if statements are executed; otherwise, the false-block statement(s) are executed.
- **Either true-block or false-block will be executed, not both.**

Nested if else Statement

- When a series of decisions are involved, we may have to use more than one if...else statement in nested form
- **General form:**

```
if (test condition-1)
{
        if (test condition-2)
        {
        Statement-1;
        }
        else
        {
        Statement-2;
        }
}
else
{
Statement-3;
}
Statement-x;
```



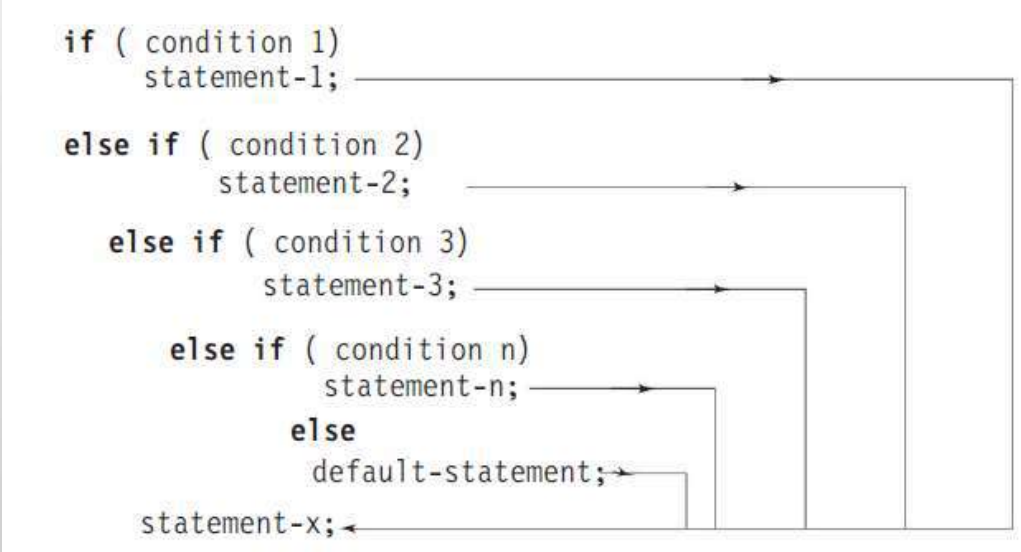
- If **condition-1** is **false**, **statement-3** will be **executed**; **otherwise** it **continues to perform the second test.**
- If **condition-2** is **true**, statement-1 will be evaluated; otherwise the **statement-2** will be **evaluated** and the **control transferred to statemet-x.**

Dangling else Problem

- One of the **classic problems encountered** when we start using **nested if….else statements** is the **dangling else**.
- Occurs **when a matching else is not available for an if**.
- Answer to this problem is very simple.
- **Always match an else to the most recent unmatched if in the current block.**
- In some cases, it is possible that the false condition is not required.
- In such situations, else statement may be omitted.
- ***Note: "else is always paired with the most recent unpaired if"***

The Else-If Ladder

- Another way of putting ***if***s together when multipath decisions are involved.
- A multipath decision is a chain of ifs in which the statement associated with each ***else*** is an ***if***
- This construct is known as the ***else if*** ladder.
- **Conditions are evaluated from the top (of the ladder), *downwards*.**
- As soon as a **true condition is found**, ***statement associated with it is executed and the control is transferred to the statement-x (skipping the rest of the ladder).***

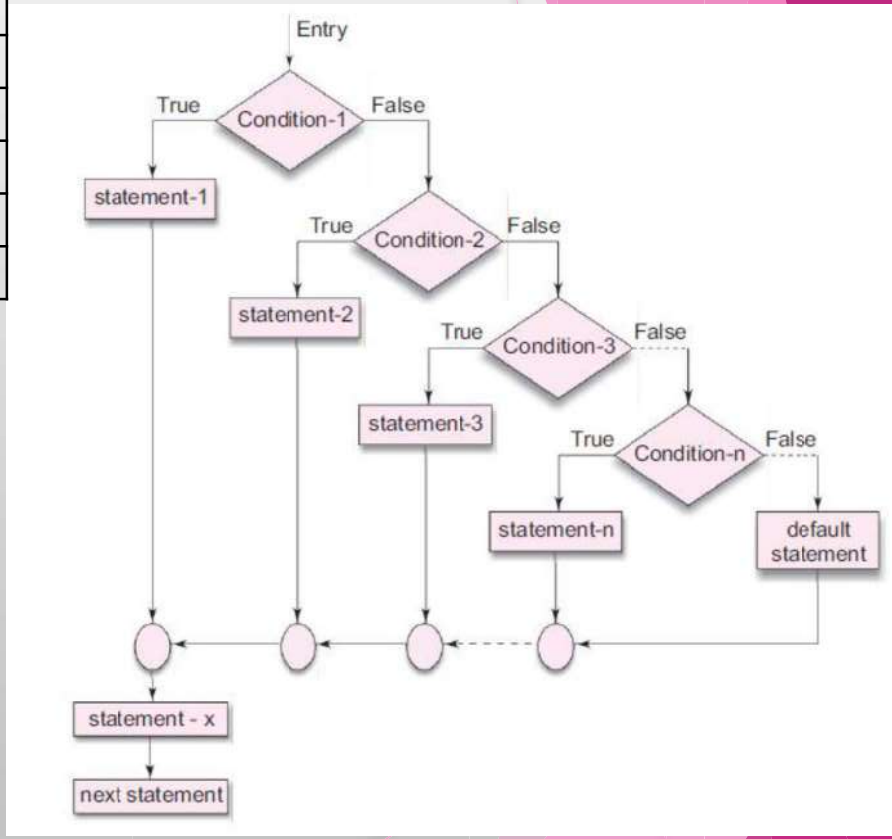


- When all the ***n conditions become false***, then the final ***else*** containing the default-statement will be executed

The Else-If Ladder contd.

- Consider an example of grading the students in an academic institution. The grading is done according to the following rules:

Average Marks	Grade
80 to 100	Honours
60 to 79	First Division
50 to 59	Second Division
40 to 49	Third Division
0 to 39	Fail



- Grading can be done using ***else if*** ladder as:

```
if(marks>79)
  grade= "Honours";
else if(marks>59)
  grade= "First Division";
else if(marks>49)
  grade= "Second Division";
else if(marks>39)
  grade= "Third Division";
else
  grade= "Fail";
printf("%s\n",grade);
```

Rules for Indentation

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

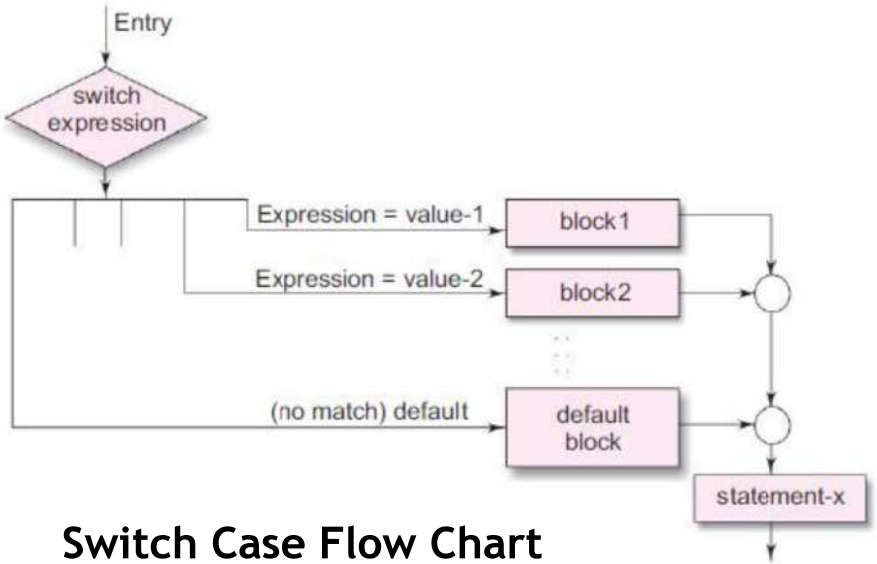
The *SWITCH* Statement

- When one of the many alternatives is to be selected, we can use an ***if*** statement to control the selection.
- The ***complexity (from readability and understandability perspective)*** of such a program increases dramatically when number of alternatives increases. C has a built-in multiway decision statement known as a ***switch***.
- ***Switch statement tests the value of a given variable (or expression) against a list of case values*** and ***when a match is found, a block of statements associated with that case is executed***.
- ***expression*** is an ***integer expression or characters***.
- ***Value-1, value-2 .....*** are constants or constant expressions (evaluable to an integral constant) and are known as ***case labels***.
- Each of these values should be unique within a switch statement.
- ***block-1, block-2 ....*** are statement lists and may contain zero or more statements. There is no need to put braces around these blocks.
- ***Note: Case labels end with a colon (:)***
- When the switch is executed, the value of the expression is successfully compared against the values ***value-1, value-2,....***
- If a case is found whose value matches with the value of the expression, then statement block that follows is executed.

```
switch (expression)
{
  case value-1:
              block-1
              break;
  case value-2:
              block-2
              break;
  ......
  ......
  default:
              default-block
              break;
}
statement-x;
```

The *SWITCH* Statement contd.

- ***Break*** statement at the end of each block signals the end of a particular case and causes an exit from the switch statement, transferring the control to the ***statement-x*** following the switch.
- ***Default*** is an optional case. When present, it will be executed if the value of the expression does not match with any of the case values. If not present, no action takes place if all matches fail and the control goes to the ***statement-x***. ***Note: ANSI C permits the use of as many as 257 case labels***.



Switch Case Flow Chart

```
---
---
index = marks/10
switch (index)
{
   case 10:
   case 9:
   case 8:
            grade = "Honours";
            break;
   case 7:
   case 6:
            grade = "First Division";
            break;
   case 5:
            grade = "Second Division";
            break;
   case 4:
            grade = "Third Division";
            break;
   default:
            grade = "Fail";
            break;
}
printf("%s\n", grade);
---
---
```

The *SWITCH* Statement contd.

```
int x,y,z;
char a,b;
float f;
```

Valid Switch	Invalid Switch	Valid Case	Invalid Case
switch(x)	switch(f)	case 3;	case 2.5;
switch(x>y)	switch(x+2.5)	case 'a';	case x;
switch(a+b-2)		case 1+2;	case x+2;
switch(func(x,y))		case 'x'>'y';	case 1,2,3;

The *SWITCH* Statement contd.

- ***Example of switch statement:***

```
#include<stdio.h>
int main(){
int number=0;printf("enter a number:");scanf("%d",&number);
switch(number){
case 10:
printf("number is equals to 10");
break;
case 50:
printf("number is equal to 50");
break;
case 100:
printf("number is equal to 100");
break;
default:
printf("number is not equal to 10, 50 or 100");
}
return 0;
}
```

The *IF-Else vs Switch Case* Statement

	If-else	switch
Definition	Depending on the condition in the 'if' statement, 'if' and 'else' blocks are executed.	The user will decide which statement is to be executed.
Expression	It contains either logical or equality expression.	It contains a single expression which can be either a character or integer variable.
Evaluation	It evaluates all types of data, such as integer, floating-point, character or Boolean.	It evaluates either an integer, or character.
Sequence of execution	First, the condition is checked. If the condition is true then 'if' block is executed otherwise 'else' block	It executes one case after another till the break keyword is not found, or the default statement is executed.
Default execution	If the condition is not true, then by default, else block will be executed.	If the value does not match with any case, then by default, default statement is executed.
Editing	Editing is not easy in the 'if-else' statement.	Cases in a switch statement are easy to maintain and modify. Therefore, we can say that the removal or editing of any case will not interrupt the execution of other cases.
Speed	If there are multiple choices implementec through 'if-else', then the speed of the execution will be slow.	If we have multiple choices then the switch statement is the best option as the speed of the execution will be much higher than 'if-else'.

Rules for *SWITCH* Statement

Rules for switch statement

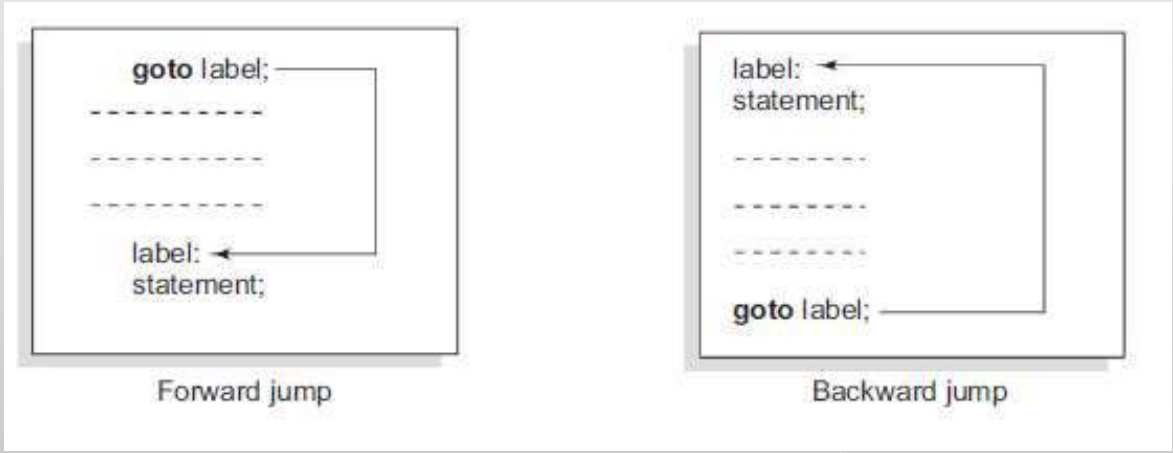
- The **switch** expression must be an integral type.
- Case labels must be constants or constant expressions.
- Case labels must be unique. No two labels can have the same value.
- Case labels must end with colon.
- The **break** statement transfers the control out of the **switch** statement.
- The **break** statement is optional. That is, two or more case labels may belong to the same statements.
- The **default** label is optional. If present, it will be executed when the ex-pression does not find a matching case label.
- There can be at most one **default** label.
- The **default** may be placed anywhere but usually placed at the end.
- It is permitted to nest **switch** statements.

The *GOTO* Statement

- C supports ***goto*** statement to branch unconditionally from one point to another in the program.
- Although it may not be essential to use the goto statement in a highly structured language like C, there may be occasions when the use of goto might be desirable.
- ***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***.
- ***label*** is placed immediately before the statement where the control is to be transferred.
- ***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;*** is encountered, program flow jump to the statement immediately following the label ***begin: and this happens unconditionally.***
- A ***goto*** is often used at the end of a program to direct the control to go to the input statement, to read further data.
- ***Note: a goto breaks the normal sequential execution of the program.***
- We should try to avoid using goto as far as possible.
- But there is nothing wrong, if we use it to enhance the readability of the program or to improve the execution speed.

The *GOTO* Statement contd.

- If the *label:* is before ***goto label;*** statement 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 ***the jump is known as a forward jump.***



The *GOTO* Statement contd.

- Consider the following example to evaluate square root of a series of numbers read from the terminal:

```
main()
{
double x,y;
read:
scanf("%f",&x);
if(x<0) goto read;
y=sqrt(x);
printf("%f%f\n",x,y);
goto read;
}
```

- The program uses two goto statements, ***one at the end, after printing the results to transfer the control back to the input statement*** and ***other to skip any further computation when the number is negative***.
- Due to the unconditional goto statement at the end, the control is always transferred back to the input statement, and subsequently pushes the computer in a permanent loop known as an ***infinite loop***.

The *GOTO* Statement contd.

- To eliminate infinite loop consider the following code:

```
#include<stdio.h>
#include<math.h>
main()
{
double x,y; int count;
count=1;
printf("Enter FIVE real values\n");/*To prevent the infinite loop*/
read:
scanf("%lf",&x); printf("\n");
if(x<0)
{   printf("Value-%d is negative\n",count);}
else
{   y=sqrt(x); printf("%f%f\n",x,y);}
count=count+1;
if(count<=5)
{   goto read; printf("\nEnd of Computation");}
}
```

Just Remember

Just Remember

- Be aware of dangling **else** statements.
- Be aware of any side effects in the control expression such as if(x++).
- Use braces to encapsulate the statements in **if** and **else** clauses of an if.... else statement.
- Check the use of =operator in place of the equal operator = =.
- Do not give any spaces between the two symbols of relational operators = =, !=, >= and <=.
- Writing !=, >= and <= operators like =!, => and =< is an error.
- Remember to use two ampersands (&&) and two bars (||) for logical operators. Use of single operators will result in logical errors.
- Do not forget to place parentheses for the if expression.
- It is an error to place a semicolon after the if expression.
- Do not use the equal operator to compare two floating-point values. They are seldom exactly equal.
- Do not forget to use a break statement when the cases in a switch statement are exclusive.
- Although it is optional, it is a good programming practice to use the default clause in a switch statement.
- It is an error to use a variable as the value in a case label of a switch statement. (Only integral constants are allowed.)
- Do not use the same constant in two case labels in a switch statement.
- Avoid using operands that have side effects in a logical binary expression such as (x--&&++y). The second operand may not be evaluated at all.
- Try to use simple logical expressions.

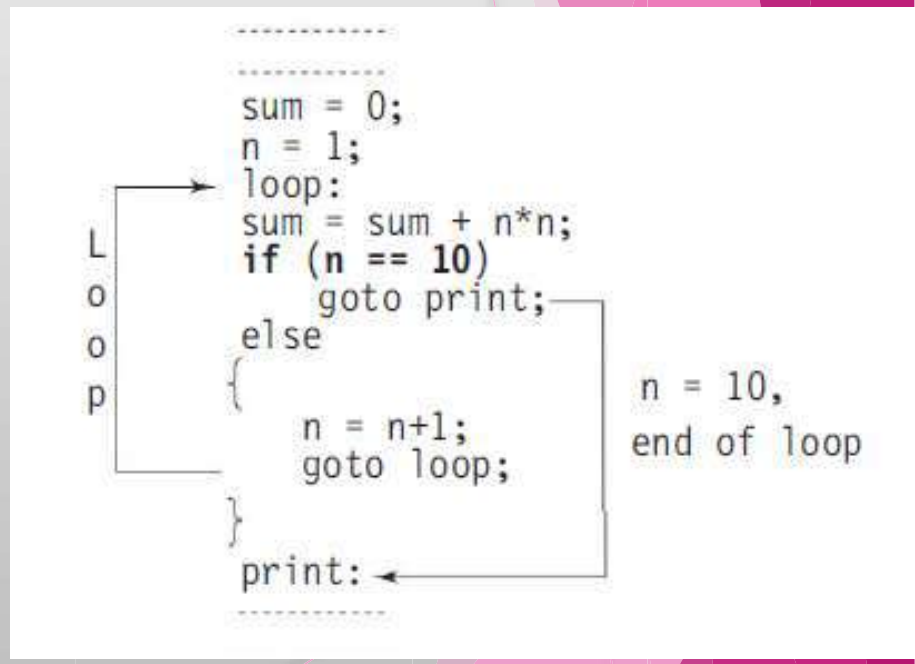
Decision Making and Looping

- It is possible to execute a segment of a program repeatedly by introducing a counter and later testing it using the if statement.
- While this method is quite satisfactory for all practical purposes, we need to initialize and increment a counter and test its value at an appropriate place in the program for the completion of the loop.
- This program does the following things:

1. Initializes the variable n.
2. Computes the square of n and adds it to sum.
3. Tests the value of n to see whether it is equal to 10 or not.

If it is equal to 10, then the program prints the results.

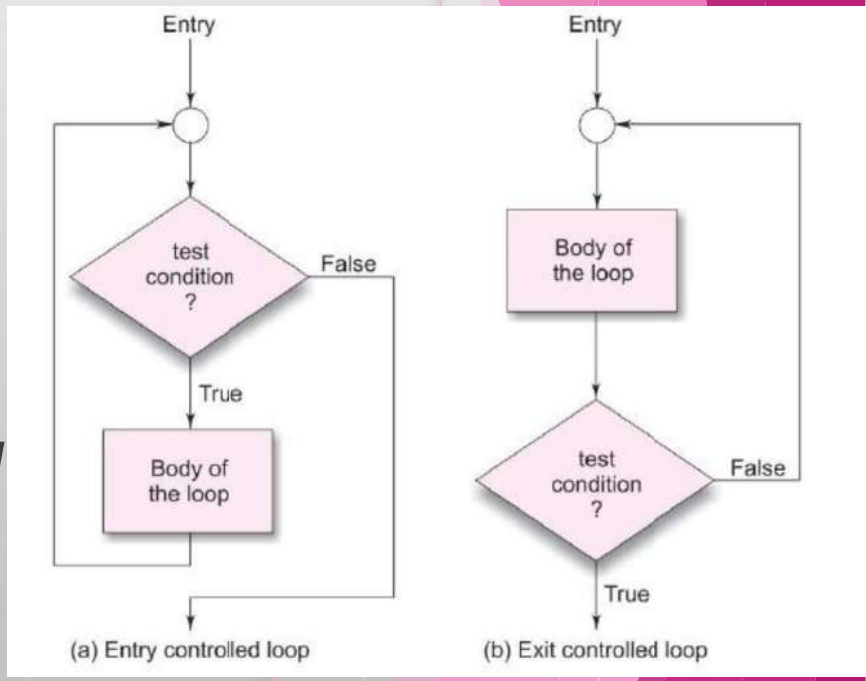
4. If n is less than 10, then it is incremented by one and the control goes back to compute the sum again.



- On occasions where the exact number of repetitions are known, there are more convenient methods of looping in C.
- The looping capabilities enable us to develop concise programs containing repetitive processes without using goto statements.

Decision Making and Looping contd.

- In looping, a sequence of statements are executed until some conditions for termination of the loop are satisfied
- A ***program loop*** consists of two segments, ***one known as the body of the loop*** and ***other known as the control statement***.
- ***Control statement tests certain conditions*** and then ***directs repeated execution of the statements contained in loop's body***.
- ***Depending on control statement position in loop***, a ***control structure*** classified either as the ***entry-controlled loop*** or as the ***exit-controlled loop***.
- In ***entry-controlled loop***, the ***control conditions are tested before the start of loop execution***. If conditions are ***not satisfied*** the ***loops body will not be executed***.
- In ***exit-controlled loop***, the ***test is performed at the end of the body*** of ***the loop*** and ***therefore loop's body is executed unconditionally for the first time***.
- ***Entry-controlled*** and ***Exit-controlled loops*** are known as ***pre-test*** and ***post-test loops*** respectively.



(a) Entry controlled loop (b) Exit controlled loop

Decision Making and Looping contd.

- Steps involved in looping procedure

1. Setting and initialization of a condition variable.
2. Execution of the statements in the loop.
3. Test for a specific value of the condition variable for execution of the loop
4. Incrementing or updating the condition variable.

- The C language provides for three constructs for performing loop operations.

1. The ***while statement***
2. The ***do while statement***
3. The ***for statement***

- Based on the nature of control variable and the kind of value assigned to it for testing the control expression, the loops are classified into two general categories: ***Counter-controlled loops and Sentinel controlled loops***
- When we know exactly how many times the loop will be executed, we use a ***counter-controlled*** loop
- We use ***control variable (counter)***. ***Counter must be initialized, tested and updated*** properly for the desired loop operations. ***Counter-controlled loop is called definite repetition loop***.
- In ***sentinel-controlled loop***, a special value ***(sentinel value)*** is used to change the loop control expression from ***true to false***. For example, when reading data we may indicate the "***end of data***" by a special value, like -1 and 999.
- The control variable is called sentinel variable. ***Sentinel-controlled loop*** is often called ***infinite repetition loop*** because iteration number is not known before the loop execution.

The *while* Statement

- Simplest of all the looping structures in C is the ***while*** statement.
- ***While*** is an entry-controlled loop statement. The *test-condition* is evaluated and if the condition is true then the body of the loop is executed.
- ***After execution, test-condition is evaluated again. If true loop body is executed.***
- The process of repeated execution of the body continues until the test-condition finally becomes false and the control is transferred out of the loop.
- Syntax:

```
while(test condition)
{
    body of the loop
}
```

- The test conditions may have compound relations as well. For instance, the statement

```
while (number > 0 && number < 100);
```

- Here the loop to be executed as long as the number keyed in lies between 0 and 100.

The *do* Statement

- The while loop construct, makes a test of condition *before* loop execution. Therefore, loop's body may not be executed at all if the condition is not satisfied at the very first attempt.
- Some occasions it is necessary to execute loop's body before test is performed, in these cases the ***do*** statement is useful.
- Syntax:

```
do
{
    body of the loop
}
while(test condition);
```

- On reaching the do statement, the program proceeds to evaluate loop's body, and at the end of the loop, the test-condition in the while statement is evaluated.
- If the condition is true, the program continues to evaluate the body of the loop once again. This process continues as long as the condition is true.
- When the condition becomes false, the loop will be terminated and the control goes to the statement that appears immediately after the while statement.
- ***Note: Since the test-condition is evaluated at the bottom of the loop, the do...while construct provides an exit-controlled loop and therefore the body of the loop is always executed at least once.***

The *for* Statement

- ***for*** loop is another entry-controlled loop that provides a more concise loop control structure.
- Syntax:

```
for(initialization;test-condition;increment)
{
    body of the loop
}
```

- The execution of the for statement is as follows:

1. ***Initialization of control variables***, using ***assignment statement*** such as ***i=1 and count = 0***. The variables ***i*** and ***count*** are called ***loop-control variables***.
2. The ***control variable value*** is tested using ***test-condition***. ***Test-condition*** is a ***relational expression***, such as ***i < 10*** that ***determines when the loop will exit***. If ***condition is true***, ***loop's body is executed***; otherwise the loop is terminated and the execution continues with the statement that immediately follows the loop.
3. When loop's body is executed, the ***control*** is ***transferred back*** to the ***for statement*** after ***evaluating last statement***. Now, the ***control variable is incremented using an assignment statement such as i = i+1*** and the ***new value of the control variable is again tested to see whether it satisfies the loop condition***. If ***condition is satisfied***, the ***loop's body is again executed***. This ***process continues till the value of the control variable fails to satisfy the test-condition***.

Note: C99 enhances the for loop by allowing declaration of variables in the initialization permits portion.

The *for* Statement contd.

- ***for*** statement allows for negative increments. ***e.g.***

```
for ( x = 9 ; x >= 0 ; x = x-1 )
{printf("%d", x);}
printf("\n");
```

- The for loop in C has several capabilities that are not found in other loop constructs. For example, more than one variable can be initialized at a time in the for statement. The statements:

```
p = 1;
for (n=0; n<17; ++n)
```

can be rewritten as

```
for (p=1, n=0; n<17; ++n)
```

- Note that the initialization section has two parts p = 1 and n = 1 separated by a comma.
- Like the initialization section, the increment section may also have more than one part. For example,

```
for (n=1, m=50; n<=m; n=n+1, m=m-1)
{p = m/n;printf("%d %d %d\n", n, m, p);}
```

is perfectly valid. The multiple arguments in the increment section are separated by commas.

- The third feature is that the test-condition may have any compound relation and the testing need not be limited only to the loop control variable

The *for* Statement contd.

- The third feature is that the test-condition may have any compound relation and the testing need not be limited only to the loop control variable. e.g.

```
sum = 0;
for (i = 1; i < 20 && sum < 100; ++i)
{
sum = sum+i;
printf("%d %d\n", i, sum);
}
```

- ***loop uses a compound test condition with the counter variable i and sentinel variable sum.***
- The loop is executed as long as both the conditions ***i < 20 and sum < 100 are true***. The sum is evaluated inside the loop.
- It is also permissible to use expressions in the assignment statements of initialization and increment sections. e.g.

```
for (x = (m+n)/2; x > 0; x = x/2)
```

is perfectly valid.

Rules for Loop Selection

Selecting a Loop

Given a problem, the programmer's first concern is to decide the type of loop structure to be used. To choose one of the three loop supported by C, we may use the following strategy:

- Analyse the problem and see whether it required a pre-test or post-test loop.
- If it requires a post-test loop, then we can use only one loop, **do while**.
- If it requires a pre-test loop, then we have two choices: **for** and **while**.
- Decide whether the loop termination requires counter-based control or sentinel-based control.
- Use **for** loop if the counter-based control is necessary.
- Use **while** loop if the sentinel-based control is required.
- Note that both the counter-controlled and sentinel-controlled loops can be implemented by all the three control structures.

The *for* Statement contd.

- Practice problems for looping.