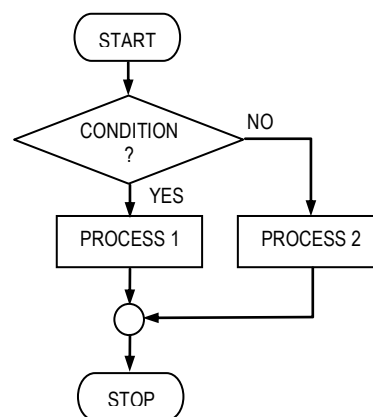


C-03 Decision Structures in C

3.1 General

In a program whenever there are more than one possible outputs depending upon some condition(s) for a particular set of input(s), we get what is called program **branching**. The control branches to different segments executing different sets of codes for each possible situation. Such a situation is shown by the flowchart to the right. Depending upon whether the condition is True (Yes) or False (No) the program branches to the process1 or process2. More than one branching is also possible. Such a situation is handled in C using the **if** & **if-else** statements and the **conditional operator** (?).



3.2 The Relational Operators:

To test or verify the conditions C provides us with some special operators along with the conventional ones as given below. The relational operators allow us to **compare** two values and find out whether one value is Equal, Not Equal, Greater Than, Less Than etc. to another value.

Equality Operator: x==y	The expression evaluates to True or 1 if x & y <u>have the same value</u> . Do not confuse this with the assignment operator, where x=y implies the value of y is assigned to x. Thus if x=3 and y=5 then x==y evaluates to False or 0 since x and y are not equal, while x=y results in x been assigned the value of y i.e. 5 and x becomes equal to 5. Thus after the operation x=y, x=y=5 whereas after the operation x==y, x and y retain their old values of 3 and 5 respectively.
Not Equal Operator: x!=y	The expression evaluates to True or 1 if x & y <u>do not have</u> the same value. Thus for x=3 and y=5, x!=y evaluates to True or 1, while for x=4 and y=4, x!=y evaluates to False or 0.
Less Than Operator: x<y	The expression evaluates to True or 1 if x is less than y else it evaluates to False or 0. Thus for x=3 and y=5, x<y evaluates to True or 1, while for x=5 and y=4, x<y evaluates to False or 0.
Greater Than Operator: x>y	The expression evaluates to True or 1 if x is greater than y else it evaluates to False or 0. Thus for x=5 and y=3, x>y evaluates to True or 1, while for x=4 and y=5, x>y evaluates to False or 0.
Less Than Equal Operator: x<=y	The expression evaluates to True or 1 if x is less than equal to y else it evaluates to False or 0. Thus for x=3 and y=5, x<=y evaluates to True or 1. Also for x=5 and y=5, x<=y evaluates to True or 1.
Greater Than Equal Operator: x>=y	The expression evaluates to True or 1 if x is greater than y else it evaluates to False or 0. Thus for x=5 and y=3, x>y evaluates to True or 1, while for x=4 and y=5, x>y evaluates to False or 0

3.3 The if-else statement:

The **if** statement tells the compiler that the instructions to follow is a decision. The general structure of the **if-else** construct is as given below:

```
If (condition is true)
{ execute statement set1;
  ... ..
}
else
{ execute statement set2;
  ... ..
}
```

Annotations:

- ← No semicolon or **then** after condition
- ← Semicolons after the statements
- ← No semicolon after **else**

After the **if** keyword the condition is placed inside brackets. If the condition evaluates true or 1 then the block of statement set1, put within the curly braces gets executed. Else if the condition evaluates false then the block of program segment set2 gets executed. Statement sets 1 & 2 can consist of one or more statements as per program requirement. Please note that there are **no semicolons (;) after the condition**. The semicolon comes after the first statement to execute after the condition. There are several variations of the above construct.

First, if there are no statements to execute in case the condition is not satisfied then the **else** statement is dropped as shown below. Only the statements inside the curly brackets get executed in case the condition is true. Else the control passes to the statement immediately after the end of the curly brackets.

If (condition is true)

```
{ execute statement set;  
  ... ..  
}
```

Program control goes here, skipping statement set, *if* condition is **NOT** true

Second, if there is only one statement to execute if the condition is satisfied, there is no need to put the braces and the statement can be put beside the condition, or to the next line. This is because the statement immediate to the condition after the **if** keyword is executed by default, if the condition is true. Only to execute multiple statements after the **if** statement we put them inside a pair of curly brackets.

If (condition is true) execute statement;

Let us now examine several examples to illustrate the above variations.

Ex.1: The program below inputs a number and prints if the number is negative.

```
main()  
{ int num;  
  printf("\nEnter any integer: ");  
  scanf("%d", &num);  
  if(num<0) printf("\nNumber is negative");  
  return(0);  
}
```

Single statement executed **if** condition is true

In the above program, the condition (**num<0**) to check if the number is negative is placed within the brackets after the **if** keyword. If the condition is satisfied then the statement following it is executed and the program prints, "Number is negative". If the number is a positive one then the condition **num<0** is not satisfied and the statement after the condition is not executed. Instead the program directly jumps to the next statement i.e. **return(0)**.

Ex.2: Now if someone wants to check the number to be positive or negative based on the input, then one can modify the above program in the following way using the **else** statement:

```
{ int num;  
  printf("\nEnter any integer: ");  
  scanf("%d", &num);  
  if(num<0)  
    printf("\nNumber is negative");  
  else  
    printf("\nNumber is positive");  
  return(0);  
}
```

Else statement executed **if** condition is *not* true

Here, when someone enters a positive number then the condition after the **if** statement is not satisfied and the **else** statement gets executed and the program prints "**Number is positive**".

Ex.3: The program can be further modified to check if the entered number is zero or not. Since a zero is not a negative number and is neither positive nor negative, we have to include another condition to check the same. There are two ways to do the same as shown below:

Method1:

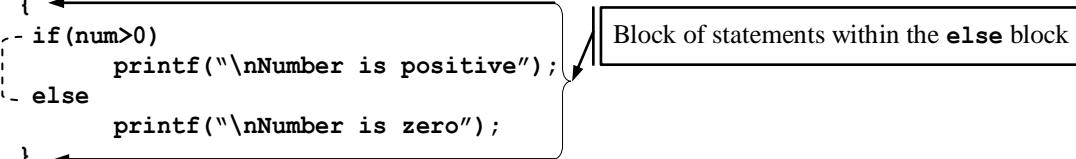
```
{ int num;  
  printf("\nEnter any integer: "); scanf("%d", &num);  
  if(num<0)  
    printf("\nNumber is negative");  
  if(num>0)  
    printf("\nNumber is positive");  
  if(num==0)  
    printf("\nNumber is zero");  
  return(0);  
}
```

Note the use of the equality operator '**==**' to **compare** the input *num* with 0.

Single statement executed if condition is true and hence not put inside curly brackets.

Method2:

```
{ int num;
  printf("\nEnter any integer: "); scanf("%d", &num);
  if(num<0)
    printf("\nNumber is negative");
  else
  {
    if(num>0)
      printf("\nNumber is positive");
    else
      printf("\nNumber is zero");
  }
  return(0);
}
```



In **Method1**, three **if** conditions are used to test the three possibilities and the outputs are printed accordingly.

In **Method2**, we have used one main **if-else** block. Within the **else** block we have placed another **if-else** block. This is known as *nesting of if-else blocks* i.e. placing one **if-else** block within another. Here the **else** statement is a multiple statement block and hence the statements to execute if the condition **num<0** is not satisfied are placed within parenthesis. The first **if** checks if the number is negative. If not, the **else** statement is executed. (If a number is not negative, then it can be either positive or zero, and this is then tested within the **else**). Within the first **else** statement it is again checked if **num>0**. If not, the only option left is that the number is equal to zero and hence no extra condition is tested and the second **else** directly prints the result i.e. "Number is zero".

The above example shows us how multiple statements can be executed within one conditional block and the concept of *nesting* one block of code within another. Certain situations may require multiple nesting i.e. putting several blocks of **if-else** one within the other as shown below:

Ex.4: The next program finds the number of digits in an integer input by the user using nested **if-else**.

```
{ int num;
  printf("\nEnter any integer (<=4 digits): "); scanf("%d", &num);
  if(num/1000)
    printf("\nA 4 digit number");
  else /*Start First else block*/
  { if(num/100)
    { printf("\nA 3 digit number");
      else /*Start Second else block*/
      { if(num/10)
        { printf("\nA 2 digit number");
          else /*Third else executes a single line*/
            printf("\nA 1 digit number");
        }
      }
    }
  }
  return(0);
}
```

The first thing to note in the above program is, **no relational operator** has been used to check the conditions. At first glance it may seem to be a bug. But let us first see what is the logic we have used to count the number of digits. We have taken the input number **num** as an *integer*. Thus:

- For a 4 digit number e.g. 2456: 2456/1000 evaluates to 2 which is a non-zero number and taken as **true**
- For a 3 digit number e.g. 169: 169/1000 evaluates to 0 (**num** being an **int**), and is taken as **false**
169/100 evaluates to 1, which is a non-zero number and taken as **true**
- For a 2 digit number e.g. 38: 38/1000 evaluates to 0 (**num** being an **int**), and is taken as **false**
38/100 evaluates to 0 (**num** being an **int**), and is taken as **false**
38/10 evaluates to 3 which is a non-zero number and taken as **true**

NOTE: Any expression which evaluates to a **non-zero positive or negative** value (e.g. 1, 230, 3.92, -210, -8.3 etc.) is taken to be **True or 1** and **only a '0' value is taken to be False** in C. Hence if a=6, b=9:

- **if (a)** → Condition is **true** or evaluates to 1 as a = 6 = (non-zero positive) = 1
- **if (-(b-a))** → Condition is **true** or evaluates to 1 as $-(b-a) = (-3) = (\text{non-zero negative}) = 1$
- **if (!a)** → Condition is **false** or evaluates to 0 as $!a = !6 = !(\text{non-zero positive}) = !1 = 0$
- **if (!(a-6))** → Condition is **true** or evaluates to 1 as $!(a-6) = !(6-6) = !0 = 1$
- **if (!(b-a))** → Condition is **false** or evaluates to 0 as $!(b-a) = !(-3) = !(\text{non-zero negative}) = !1 = 0$
- **if (-!a)** → Condition is **false** or evaluates to 0 as $-!a = -!6 = -!(\text{non-zero positive}) = -!1 = 0$
- **if (!(a>b))** → Condition is **true** or evaluates to 1 as $!(a>b) = !(6>9) = !(False) = True = 1$
- **if (!a==b)** → Condition is **false** or evaluates to 0 as $!a==b = (!6==9) = (0==9) = False = 0$
- **if (!(a==b))** → Condition is **true** or evaluates to 1 as $!(a==b) = !(6==9) = !(False) = True = 1$
- **if (!a==!b)** → Condition is **true** or evaluates to 1 as $!a==!b = (!6==!7) = (0==0) = True = 1$

Thus in **Ex4**, the values of the expressions evaluated within the conditions are *sufficient* to make any decision and it is redundant to write for example **(num/100>0)**. In the first case **num/1000** evaluates to 2, which is a non-zero number and hence is treated as true. The condition being satisfied (i.e. true) the **if** part is executed, the **else** part is neglected and the program will print **"A 4 digit number"**.

Suppose now if we input 38 i.e. a two digit number, then the first **if** condition i.e. **(38/1000=0)** will evaluate false and the control will enter the **else** part. Inside the **else** part is another **if** block which also evaluates false as **(38/100=0)** and the control is made to enter the second **else** block. There the third **if** condition is satisfied and the program prints **"A 2 digit number"**. For a single digit number, none of the **if** conditions are satisfied and hence the third and last **else** statement gets executed and the program prints **"A 1 digit number"**.

Here we have used multiple nesting of **if-else** statements to find the number of digits in a given integer number. There are other ways of doing the same thing using loops, that we will discuss later.

Ex.5: The next program shows the use of multiple statements within each **if-else** block. The program inputs the total purchase value of a customer and calculates the net amount payable by the customer after offering different discount values depending upon the amount.

```
main()
{ float amt, total;
  printf("\nEnter the total purchase amount (Rs.): ");
  scanf("%f", &amt);
  if(amt<2000.0)
  { total=0.9*amt;
    printf("\n10 percent discount is provided");
    printf("\nThe net amount payable = Rs. %.2f", total);
  }
  else
  { total=0.9*2000.0 + 0.8*(amt-2000.0);
    printf("\n10 percent discount on Rs2000 = Rs%.2f",0.9*2000);
    printf("\n20 percent discount on Rs%.2f = Rs%.2f",amt-2000,0.8*(amt-2000));
    printf("\nThe net amount payable = Rs%.2f", total);
  }
  return(0);
}
```

Points to note while using if-else:

Ex6: The following program portion checks a number to be even or odd, and if the number is even then checks, if it is a perfect square.

```
{ int root, num;
  printf("\nEnter a number: "); scanf("%d", &num);
  root=sqrt(num);
  if(num%2==0) /*first if statement*/
    if(num-root*root==0) /*second if statement*/
      printf("\nNumber is even and a perfect square");
  else
    printf("\nNumber is odd");
}
```

At a first glance it may seem that if the entered number is an even perfect square then the program will print accordingly and if not it will print "Number is odd". In reality if an odd number is entered the program will print nothing. The bug lies in the use of the **else** statement. From the program indentation it may seem that the **else** goes with the first **if**. In reality it goes with the second **if**. The general rule is that **else goes with the nearest if**. To make the else statement execute, we will have to place the second **if** and the following **printf()** statement within curly brackets as shown below.

```
main()
{ int root, num;
  printf("\nEnter a number: "); scanf("%d", &num);
  root=sqrt(num);

  if(num%2==0)
  {if(num-root*root==0)
   printf("\nNumber is even and a perfect square");
  }
  else
   printf("\nNumber is odd");
  return 0;
}
```

Ex7: The following program is a guessing game, which initialises a variable and asks the user to guess a number. Then depending upon a pre-written condition and the input, executes a certain portion of the code.

```
main()
{ int product, num, val=10000;
  printf("\nGuess a number between 1 and 9: "); scanf("%d", &num);
  product=val*num;
  if(product>30000) printf("\nYou have won!!");
  else printf("\nYou have lost at last!! Better luck next time");
  return(0);
}
```

The program is expected to print "You have won!!", in case someone types values from 4 to 9 (as product will be $10000 \times 4 = 40000 > 30000$ etc. and hence the condition will evaluate to true), and print "You have lost at last!! Better luck next time", if someone types 1 to 3.

But irrespective of the input, the output will be always "You have lost at last!! Better luck next time". Guess why! If someone types say 5, then **product**= $5 \times 10000 = 50000$. But **product** has been defined as an **int** and hence if its value crosses +32767, it will cross its range and assume a value on the negative side and hence the value will be less than 30000. Thus the condition will always become false and execute the **else** statement. To rectify such problems, *the program variables should be properly chosen keeping in mind the range of values they can take in the course of running the program*. In this case the variable **product** has to be defined as a **long int** to ensure proper running of the program.

Ex8: This program inputs the initials of a person and based on a condition executes certain portion of the code.

```
main()
{ char initial_1, initial_2;
  printf("\nEnter your first initial: "); initial_1=getchar();
  printf("\nEnter your second initial: "); initial_2=getchar();
  if(initial_1>initial_2)
   printf("\nYour name has a higher alphabetical order than your surname");
  else
   printf("\nYour surname has a higher alphabetical order than your name");
  return(0);
}
```

The above simple program inputs two characters (the initials of a person) and checks the order of the alphabets. Here the **if** condition is used to check the two characters. While checking characters, the *ASCII values* of the characters are checked to arrive at a decision. Suppose someone enters the initials 'T' and 'C'. Thus **initial_1**='T' = ASCII 84 and **initial_2**='C' = ASCII 67 and (**initial_1**>**initial_2**) evaluates to (84>67) which is True and prints the first statement.

Ex9: The following program inputs a floating-point variable and compares it with a **float literal**.

```
main()
{ float guess;
  printf("\nGuess the value of pi (upto 2 digits after decimal): ");
  scanf("%f", &guess);
  if(guess>3.14) printf("\nYour guess was on the higher side");
  if(guess==3.14) printf("\nYou have guessed correctly");
  if(guess<3.14) printf("\nYour guess was on the lower side");
  return(0);
}
```

From the logic of the above program it is evident that if someone guesses pi as 3.14 then the program should print "You have guessed correctly". But contrary to common sense, to your surprise the program will print, "Your guess was on the higher side". There is nothing wrong as far as program execution is concerned because a floating-point value is not stored exactly as it is shown on the screen but due to precision considerations it is stored as something less than 3.14. Hence the comparison makes **guess>3.14** as true and prints the statement accordingly.

To overcome such a problem, declare **guess** as a **double** and use **%lf** format specifier in the **scanf()** function to input the number. Another method is to forcibly make the literal 3.14 a float, as shown below:

```
main()
{ float guess;
  printf("\nGuess the value of pi (upto 2 digits after decimal): ");
  scanf("%f", &guess);
  if(guess>(float)3.14) printf("\nYour guess was on the higher side");
  if(guess==(float)3.14) printf("\nYou have guessed correctly");
  if(guess<(float)3.14) printf("\nYour guess was on the lower side");
  return(0);
}
```

This is known as **type casting** whereby the number **3.14** is converted *temporarily* to a specified data type by the syntax **(float)3.14**. Thus before making the comparison **3.14** will be converted from a **long double** to a **float** and hence will yield the correct result. More of casting of data types will be discussed later. Thus **be careful when comparing floating point variables within a 'if' statement.**

Ex10: The next program inputs a number and prints if the number is equal to a preset number.

```
main()
{ int num;
  printf("\nGuess a number between 1 and 9: "); scanf("%d", &num);
  if(num=6)
    printf("\nYou have entered 6");
  else
    printf("\nYou have entered %d", num);
  return(0);
}
```

Whatever be the input by the user, the above program will *always* print "You have entered 6". Reason? In the **if** condition, instead of using the equality operator (**=**) to compare the two values, the assignment operator (**=**) has been used. This results in *assigning* the value of 6 to **num** instead of comparing the same. Thus **num** always becomes equal to 6 and results in a permanent truth condition printing the first line. This is a common programming mistake and care should be taken to avoid the same.

3.4 The Logical Operators (AND, OR, NOT)

When more than one relation needs to be tested to arrive at a decision, then nested **if-else** statements can be used. Instead of using multiple levels of **if-else**, which can be at times much confusing and lengthy and giving rise to errors, one can take help of the Logical Operators. C provides us with three logical operators, which does the same logical functions in C as with Boolean functions. These are the following:

- **!** : The logical **NOT** operator
- **&&** : The logical **AND** operator
- **||** : The logical **OR** operator

Priority
↓

Of the above operators, the unary operator '!' i.e. NOT has the highest priority followed by AND, then by OR. This order helps one to correctly evaluate an expression when more than one operators are present in an expression. Let us now examine a simple example to see how the logical operators work.

Ex.11: The result of an examination is to be printed based on the marks received by the student. The total marks are 300. The program is to print "Excellent" if the marks are equal or above 80%, "Good" if marks are from 60% to 80%, "Fair" if marks are from 40% to 60% and "Poor" if the marks are below 40%.

First we write the program using **if-else** as shown below:

```
{  int marks; float percent;
  printf("\nEnter total marks of student: "); scanf("%d", &marks);
  percent=(marks/300.0)*100.0;
  if (percent>=80)
    printf("\nExcellent");
  else
    { if(percent>=60)
      printf("\nGood");
      else
        { if(percent>=40)
          printf("\nFair");
          else
            printf("\nPoor");
        }
    }
  return(0);
}
```

Now we write the same program using **logical operators**:

```
{  int marks; float percent;
  printf("\nEnter total marks of student: "); scanf("%d", &marks);
  percent=(marks/300.0)*100.0;
  if(percent>=80) printf("\nExcellent");
  if((percent>=60)&&(percent<80)) printf("\nGood");
  if((percent>=40)&&(percent<60)) printf("\nFair");
  if(percent<40) printf("\nPoor");
  return(0);
}
```

It is evident that the second method is much more compact and less prone to errors. We have used the logical operator && i.e. AND to set the range of values for getting Good and Fair. Only if the marks are above or equal to 60 and at the same time below 80, will a person get Good. Let us write down how the conditions evaluate for a marks of 55. As is evident, the first two and last *if* condition fails and the program control executes the third *if* condition only. The outputs of the four *if* conditions are given below to understand the logic of the operators.

First Condition: (55>=80) = False

Second Condition: ((55>=60)&&(55<80)) = ((False)&&(True)) = (0&&1) = 0 = False

Third condition: ((55>=40)&&(55<60)) = ((True)&&(True)) = (1&&1) = 1 = True

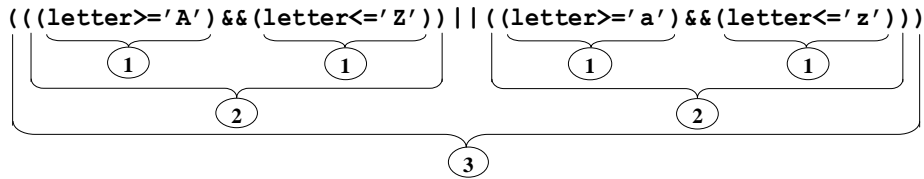
Fourth Condition: (55<40) = False

Ex.12: The user is asked to enter a character from the keyboard. If the user enters an alphabet the program prints accordingly else it says that the entered number is not an alphabet.

```
main()
{  char letter;
  printf("\nEnter any character: "); letter=getchar();
  if(((letter>='A')&&(letter<='Z'))||((letter>='a')&&(letter<='z')))
    printf("\nYou have entered an alphabet");
  else
    printf("\nYou have entered a symbol or a digit");
  return(0);
}
```

The above program uses the ASCII values of the alphabets to check the condition. (Note: the characters A to Z have ASCII values from 65 to 90 and the characters a to z have ASCII values from 97 to 122). Thus if

the ASCII value of a character is greater than equal to 65 AND less than equal 90 then it is an alphabet (Capital letter) OR if the ASCII value of a character is greater than equal to 97 AND less than equal 122 then also it is an alphabet (Small letter). The order in which the conditions are checked are shown below. First portion ① then portion ② and finally portion ③ is being executed.



Thus if someone enters 'p' the conditions evaluate as (with `letter = 'p'` = ASCII 112):

```
(( ('p' >= 'A') && ('p' <= 'Z') ) || ( ('p' >= 'a') && ('p' <= 'z') ))
= (( (112 >= 65) && (112 <= 90) ) || ( (112 >= 97) && (112 <= 122) ))
= (( (True) && (False) ) || ( (True) && (True) ))
= ( (1 && 0) || (1 && 1) )
= ( (0) || (1) )
= (1)
= True
```

Ex.13: An insurance company is to issue insurance to a driver if the following conditions are satisfied.

- If driver is married
- If driver is unmarried male and above 30 years of age
- If driver is unmarried female and above 25 years of age

In solving this problem let us first see what are the inputs that are required and specify codes for the same:

- Married or Unmarried: Let us declare an `int` variable called `ms`
where `ms=1` means married and `ms=0` means unmarried
- Male or Female: Let us declare an `int` variable called `sex`
where `sex=1` means male and `sex=0` means female
- Age: Let us declare an `int` variable called `age`

Now write the condition to issue an insurance using the above variables and logical operators.

```
((ms==1) || ((ms==0) && (sex==1) && (age>30)) || ((ms==0) && (sex==0) && (age>25)))
```

a
b
c

The three conditions a, b, c, are separated by OR operators, for if *any one* of the conditions are true then insurance can be issued. Again within each condition the sub-conditions are connected by AND operators, for all those sub-conditions have to be satisfied *simultaneously* to satisfy a condition. To avoid any confusion regarding the priority of operations, each operation is being placed within separate brackets.

The NOT (!) operator is used to reverse the logical value of a variable as shown in the example below:

Ex.14: Consider the following code where a number is tested to be multiple of another number.

```
main()
{ int num1, num2;
  printf("\nEnter larger integer: "); scanf("%d", &num1);
  printf("\nEnter smaller integer: "); scanf("%d", &num2);
  if(!(num1%num2))
    printf("\n%d is a multiple of %d", num1, num2);
  else
    printf("\n%d is not a multiple of %d", num1, num2);
  return(0);
}
```

In the above program if `num1` is a multiple of `num2` then `num1/num2` will produce no remainder and hence `num1%num2` will be 0. This condition is tested in the `if` statement. But if the relation was used alone, then the condition would have resulted in, `if (num1%num2) = if (0) = False` in case `num1` is a multiple of `num2` and would not execute the statement following it. A way out would be to *interchange* the *if-else* print statements or use the equality condition `if (num1%num2==0)`. Instead we have used the NOT operator to reverse the logical value, so that when the relation is satisfied we get, `if (!0)=if (1)=True`.

Points to note while using logical operators:

A logical expression is evaluated from left to right. In case sufficient logical truth has been obtained to evaluate an expression, then further evaluation of the expression is stopped. Thus in the case of **Ex.13**, if **ms==1**, then the first expression evaluates true and is sufficient to make the whole condition true (as the other conditions are connected by OR). Thus the evaluation of the remaining expressions is not carried out.

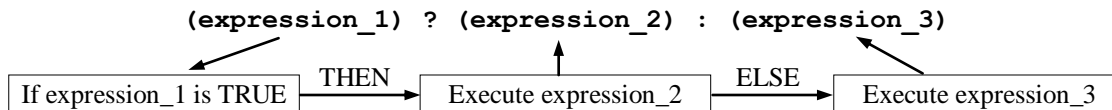
`(ms==1 || ms==0&&sex==1&&age>30 | ms==0&&sex==0&&age>25)`

If this is true

Evaluation of this is not carried out

3.5 The Conditional Operator (? :)

The conditional operator is sometimes also called the **ternary operator** as it carries **three arguments** with it. The general format of the conditional operator is as follows:



The operator is similar to the *if-else* construct. But unlike an *if-else* construct, each expression can hold only a single statement whereas each of the *if* and *else* block can hold multiple statements. Though multiple statements are not possible while using the conditional operator, but nesting of one conditional operator into another is possible as will be shown. The following examples illustrate the use of the conditional operator.

Ex15.: To find the greater of two numbers.

```
main()
{
    int a, b;
    printf("\nEnter number1: "); scanf("%d", &a);
    printf("\nEnter number2: "); scanf("%d", &b);
    (a>b)?(printf("\na is greater than b")):(printf("\nb is greater than a"));
    return(0);
}
```

In the above example, if **a** is greater than **b**, then **expression_1** is satisfied and hence **expression_2** gets executed with the program printing “a is greater than b”. If **b** is greater than **a**, then the expression_1 is not satisfied and hence expression_3 gets executed with the program printing “b is greater than a”. Here the conditional operator executes by itself based on the condition. When two numbers are evaluated as part of a calculation then the above procedure may be used to separate the higher of the two numbers, if required.

The following program is a little complex one and shows the use of nested **conditional operators**. The conditional operator evaluates to a Boolean value that is either true (1) or false (0) and the **result is assigned** to another variable **flag**.

Ex16.: To check if for any three integers a, b, c: a=b+c or b=c+a or c=a+b (This logic can also be used to check if three points are collinear, or if three numbers are Pythagorean or not). The logic of the program is:

Of the three numbers, if **a>b** then it is checked if **a>c**. If so, then **a>b** and **c**. If not, then **c>a**. i.e. **c>a>b**. Thus **c>a** and **b** both.

If **b>a** then it is checked if **b>c** also. If so then **b>a** and **c**. If not then **c>b**. i.e. **c>b>a**. Thus **c>a** and **b** both.

```
main()
{int a, b, c, flag;
 printf("\nEnter first integer: "); scanf("%d", &a);
 printf("\nEnter second integer: "); scanf("%d", &b);
 printf("\nEnter third integer: "); scanf("%d", &c);
 flag = ((a>b)?((a>c)?(a==b+c):(c==a+b)):(b>c)?(b==a+c):(c==a+b)));
 (flag==1)?(printf("\nRelation satisfied")):(printf("\nRelation not satisfied"));
 return(0);
}
```

