M. Trupthi

Asst.Prof., IT Department,

CBIT, Gandipet, Hyderabad

Introduction To C Language

A C Program is made up of statements. A Statement is a part of your program that can be executed. In other words every statement in your program alone or in combination specifies an action to perform by your program. C provides variety of statements to help you attain any function with maximum flexibility and efficiency. One of the reasons for popularity of C is because of the extreme power provided to programmer in C due to It's rich and diverse set of statements define in C. For becoming a top notch programmer you must have clear understanding of the C statements and the situations where statements in C are applicable.

if

The first form of the `if` statement is an all or nothing choice. `if` some condition is satisfied, do what is in the braces, otherwise just skip what is in the braces. Formally, this is written:

```
if (condition) statement;
```

or

```
if (condition)
   {
   compound statement
   }
```

A condition is usually some kind of comparison. It must have a value which is either true or false (1 or 0) and it must be enclosed by the parentheses `(` and `)`. If the condition has the value `true' then the statement or compound statement following the condition will be carried out, otherwise it will be ignored. Some of the following examples help to show this:

```
int i;

printf ("Type in an integer");

scanf ("%ld",&i);

if (i == 0)
   {
   printf ("The number was zero");
   }

if (i > 0)
   {
   printf ("The number was positive");
   }

if (i < 0)
   {
   printf ("The number was negative");
   }
```

The same code could be written more briefly, but perhaps less consistently in the following way:

```
int i;

printf ("Type in an integer");

scanf ("%ld",&i);

if (i == 0) printf ("The number was zero");
if (i > 0) printf ("The number was positive");
if (i < 0) printf ("The number was negative");
```

if … else

The `if .. else` statement has the form:

```
if (condition) statement1; else statement2;
```

This is most often written in the compound statement form:

```
if (condition)
   {
   statements
   }
else
   {
   statements
   }
```

The `if..else` statement is a two way branch: it means do one thing or the other. When it is executed, the condition is evaluated and if it has the value `true' (i.e. not zero) then *statement1* is executed. If the condition is `false' (or zero) then *statement2* is executed. The `if..else` construction often saves an unnecessary test from having to be made. For instance:

```
int i;

scanf ("%ld",i);

if (i > 0)
   {
   printf ("That number was positive!");
   }
else
   {
   printf ("That number was negative or zero!");
   }
```

It is not necessary to test whether `i` was negative in the second block because it was implied by the `if..else` structure. That is, that block would not have been executed unless `i` were NOT greater than zero.

Nested `if` s and logic

Consider the following statements which decide upon the value of some variable `i`. Their purposes are exactly the same.

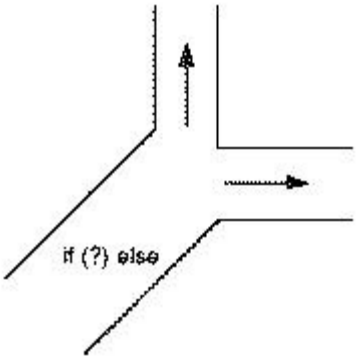
```
if ((i > 2) && (i < 4))
   {
   printf ("i is three");
   }
```

or:

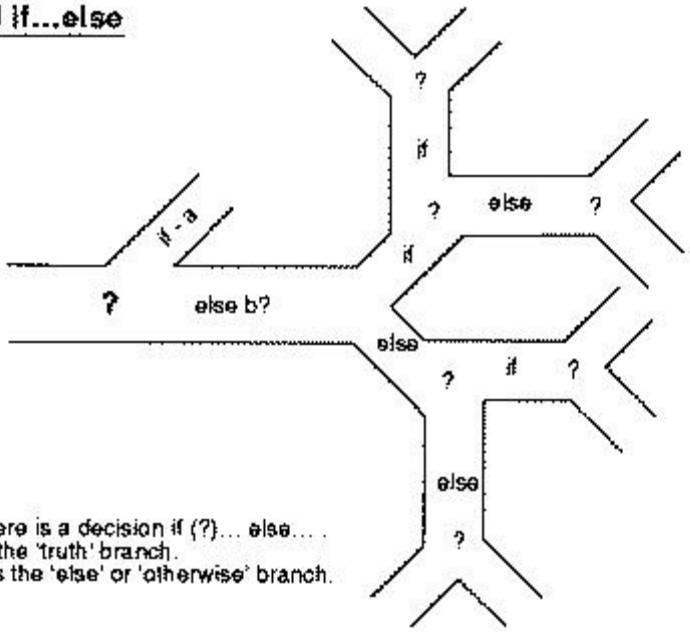
```
if (i > 2)
   {
   if (i < 4)
      {
      printf ("i is three");
      }
   }
```

Both of these test `i` for the same information, but they do it in different ways. The first method might been born out of the following sequence of thought:

if...else



Nested if...else



At each fork there is a decision if (?)... else... .
The left fork is the 'truth' branch.
The right fork is the 'else' or 'otherwise' branch.

`switch`: integers and characters

The `switch` construction is another way of making a program path branch into lots of different limbs. It works like a kind of multi-way switch. The switch statement has the following form:

```
switch (int or char expression)
   {
   case constant : statement;
                   break;          /* optional */
   ...
   }
```

It has an expression which is evaluated and a number of constant `cases' which are to be chosen from, each of which is followed by a statement or compound statement. An extra statement called `break` can also be incorporated into the block at any point. `break` is a reserved word.

The switch statement can be written more specifically for integers:

```
switch (integer value)

   {
   case 1:  statement1;
            break;                 /* optional line */

   case 2:  statement2;
            break;                 /* optional line */

   ....

   default: default statement
            break;                  /* optional line */
   }
```

When a switch statement is encountered, the expression in the parentheses is evaluated and the program checks to see whether the result of that expression matches any of the constants labeled with `case`. If a match is made (for instance, if the expression is evaluated to 23 and there is a statement beginning "case 23 : ...") execution will start just after that case statement and will carry on until either the closing brace `}` is encountered or a break statement is found. `break` is a handy way of jumping straight out of the switch block. One of the cases is called `default`. Statements which follow the `default` case are executed for all cases which are not specifically listed. `switch` is a way of choosing some action from a number of known instances. Look at the following example.

```
#include <stdio.h>

#define CODE 0

/*************************************************/
```

```
main ()

{ short digit;

printf ("Enter any digit in the range 0..9");

scanf ("%h",&digit);

if ((digit < 0) || (digit > 9))
   {
   printf ("Number was not in range 0..9");
   return (CODE);
   }

printf ("The Morse code of that digit is ");
Morse (digit);
}

/*********************************************/

Morse (digit)        /* print out Morse code */

short digit;

{
switch (digit)
   {
   case 0 : printf ("-----");
            break;
   case 1 : printf (".----");
            break;
   case 2 : printf ("..---");
            break;
   case 3 : printf ("...--");
            break;
   case 4 : printf ("....-");
            break;
   case 5 : printf (".....");
            break;
   case 6 : printf ("-....");
            break;
   case 7 : printf ("--...");
            break;
   case 8 : printf ("---..");
            break;
   case 9 : printf ("----.");
   }
}
```

The program selects one of the printf statements using a switch construction. At every `case` in the switch, a `break` statement is used. This causes control to jump straight out of the switch statement to its closing brace `}`. If break were not included it would go right on executing the statements to the end, testing the cases in turn. break this gives a way of jumping out of a switch quickly.

There might be cases where it is not necessary or not desirable to jump out of the switch immediately. Think of a function `yes()` which gets a character from the user and tests whether it was 'y' or 'Y'.

```
yes ()         /* A sloppy but simple function */

{
switch (getchar())
   {
   case 'y' :
   case 'Y' : return TRUE
   default  : return FALSE
   }
}
```

If the character is either 'y' or 'Y' then the function meets the statement `return TRUE`. If there had been a break statement after case 'y' then control would not have been able to reach case 'Y' as well. The return statement does more than break out of switch, it breaks out of the whole function, so in this case break was not required. The default option ensures that whatever else the character is, the function returns false.

Loops

Controlling repetitive processes. Nesting loops

Decisions can also be used to make up loops. They allow the programmer to build a sequence of instructions which can be executed again and again, with some condition deciding when they will stop. There are three kinds of loop in C. They are called:

- `while`
- `do` ... `while`
- `for`

These three loops offer a great amount of flexibility to programmers and can be used in some surprising ways!

`while`

The simplest of the three loops is the while loop. In common language `while` has a fairly obvious meaning: the `while`-loop has a condition:

`while (`*`condition`*`)`

```
   {
   statements;
   }
```

and the statements in the curly braces are executed while the condition has the value "true" (1).,.

The first important thing about this loop is that has a conditional expression (something like `(a > b)` etc...) which is evaluated every time the loop is executed by the computer. If the value of the expression is true, then it will carry on with the instructions in the curly braces. If the expression evaluates to `false` (or 0) then the instructions in the braces are ignored and the entire while loop ends. The computer then moves onto the next statement in the program.

The second thing to notice about this loop is that the conditional expression comes at the start of the loop: this means that the condition is tested at the start of every `pass', not at the end. The reason that this is important is this: if the condition has the value false before the loop has been executed even once, the statements inside the braces will not get executed at all - not even once.

This example listing prompts the user to type in a line of text and it counts all the spaces in that line. It quits when there is no more input left and printf out the number of spaces.

```
/*********************************************/
/*                                           */
/* while loop                                */
/*                                           */
/*********************************************/

 /* count all the spaces in an line of input */


#include <stdio.h>

main ()

{ char ch;
   short count = 0;

printf ("Type in a line of text\n");

while ((ch = getchar()) != '\n')
   {
   if (ch == ' ')
      {
      count++;
      }
   }

printf ("Number of space = %d\n",count);
}
```

do..while

The do..while loop resembles most closely the repeat..until loops of Pascal and BASIC except that it is the `logical opposite'. The do loop has the form:

```
do
   {
   statements;
   }

while (condition)
```

Notice that the condition is at the end of this loop. This means that a do..while loop will always be executed at least once, before the test is made to determine whether it should continue. This is the only difference between while and do..while.

A do..while loop is like the "repeat .. until" of other languages in the following sense: if the condition is NOTed using the ! operator, then the two are identical.

```
repeat                    do

                  ==

until(condition)             while (!condition)
```

Here is an example of the use of a do..while loop. This program gets a line of input from the user and checks whether it contains a string marked out with "" quote marks. If a string is found, the program prints out the contents of the string only. A typical input line might be:

```
    Onceupon a time "Here we go round the..."what a terrible..
```

The output would then be:

```
Here we go round the...
```

```
/*******************************************/
/*                                         */
/* do .. while demo                        */
/*                                         */
/*******************************************/

   /* print a string enclosed by quotes " " */
   /* gets input from stdin i.e. keyboard   */
   /* skips anything outside the quotes     */


#include <stdio.h>
```

```
/************************************************/
/* Level 0                                      */
/************************************************/

main ()

{ char ch,skipstring();

do
   {
   if ((ch = getchar()) == '"')
      {
      printf ("The string was:\n");
      ch = skipstring();
      }
   }

while (ch != '\n')
   {
   }
}
```

for

The most interesting and also the most difficult of all the loops is the for loop.

> For all values of *variable* from *value1* to *value2* in steps of `value3`, repeat the following sequence of commands....

In BASIC this looks like:

```
FOR variable = value1 TO value2 STEP value3

NEXT variable
```

It is actually based upon the `while` construction. A `for` loop normally has the characteristic feature of controlling one particular variable, called the control variable. That variable is somehow associated with the loop. For example it might be a variable which is used to count "for values from 0 to 10" or whatever. The form of the for loop is:

```
for (statement1; condition; statement2)
   {
   }
```

For normal usage, these expressions have the following significance.

statement1

This is some kind of expression which initializes the control variable. This statement is only carried out once before the start of the loop. e.g. `i = 0;`

condition

This is a condition which behaves like the while loop. The condition is evaluated at the beginning of every loop and the loop is only carried out while this expression is true. e.g. `i < 20;`

statement2

This is some kind of expression for altering the value of the control variable. In languages such as Pascal this always means adding or subtracting 1 from the variable. In C it can be absolutely anything. e.g. `i++` or `i *= 20` or `i /= 2.3` ...

Compare a C for loop to the BASIC for loop. Here is an example in which the loop counts from 0 to 10 in steps of 0.5:

```
FOR X = 0 TO 10 STEP 0.5

NEXT X
```

```
for (x = 0; x <= 10; x += 0.5)
   {
   }
```

example to find the sum of the first n natural numbers very simply:

```
sum = 0;

for (i = 0; i <= n; i++)
   {
   sum += i;
   }
```

It generally finds itself useful in applications where a single variable has to be controlled in a well determined way.

```
/*********************************************/
/*                                           */
/* Prime Number Generator #1                 */
/*                                           */
/*********************************************/

   /* Check for prime number by raw number */
   /* crunching. Try dividing all numbers  */
   /* up to half the size of a given i, if */
   /* remainder == 0 then not prime!       */

#include <stdio.h>

#define MAXINT  500
#define TRUE      1
#define FALSE     0

/*********************************************/
```

```
/* Level 0                                          */
/**************************************************/

main ()

{ int i;

for (i = 2; i <= MAXINT; i++)
   {
   if (prime(i))
      {
      printf ("%5d",i);
      }
   }
}

/**************************************************/
/* Level 1                                          */
/**************************************************/

prime (i)         /* check for a prime number */

int i;

{ int j;

for (j = 2; j <= i/2; j++)
   {
   if (i % j == 0)
      {
      return FALSE;
      }
   }

return TRUE;
}
```

C break and continue Statements

break statement is used to break any type of loop such as *while*, *do while* an *for* loop. *break* statement terminates the loop body immediately. *continue* statement is used to break current iteration. After ***continue*** statement the control returns to the top of the loop test conditions.

Here is an example of using *break* and *continue* statement:

```
01. #include <stdio.h>
02. #define SIZE 10
03. void main(){
04.
05.    // demonstration of using break statement
06.
07.    int items[SIZE] = {1,3,2,4,5,6,9,7,10,0};
08.
09.    int number_found = 4,i;
10.
11.    for(i = 0; i < SIZE;i++){
```

```
12.
13.        if(items[i] == number_found){
14.
15.            printf("number found at position %d\n",i);
16.
17.            break;// break the loop
18.
19.        }
20.        printf("finding at position %d\n",i);
21.    }
22.
23.    // demonstration of using continue statement
24.    for(i = 0; i < SIZE;i++){
25.
26.        if(items[i] != number_found){
27.
28.            printf("finding at position %d\n",i);
29.
30.            continue;// break current iteration
31.        }
32.
33.        // print number found and break the loop
34.
35.        printf("number found at position %d\n",i);
36.
37.        break;
38.    }
39.
40.}
```

Here is the output

```
finding at position 0
finding at position 1
finding at position 2
number found at position 3
finding at position 0
finding at position 1
finding at position 2
number found at position 3
```

goto Statement

goto statement is a jump statement and is perhaps the most controversial feature in C. Goto in unstructured language such as basic was indispensable and the soul of the programming language but in a structured language they are criticized beyond limit. The critics say that due to other jump statements in c goto is not all required and more over it destroys the structure of your program and make it unreadable. But on other hand goto statement of C provides some very powerful programming options in very complicated situation. This tutorial will take a neutral stand in such an argument and neither recommend nor discourage use of goto.

Goto is used for jumping from one point to another point in your function. You can not jump from one function to another. Jump points or way points for goto are marked by label statements. Label statement can be anywhere in the function above or below the goto statement. Special situation in which goto find use is deeply nested loops or if - else ladders.

General form of goto statement is -

```
.
.
.
goto label1;
.
.
.
.
label1 :
.
.
.
label2 :
.
.
.
.
goto label2;
```

To further clarify the concept of goto statement study the C Source Code below -

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

```
int main (void)// Print value 0 to 9
{
int a;
a = 1;
loop:;  // label stament
printf ("\n%d",a);
a++;
if (a < 10) goto loop ; // jump statement
return 0;
}
```