Chapter: 2

C Control Statements

General structures of : if, else-if, do, while, for & switch statements

2.1 The Block of statements or Code Bolcks

Statement : A statement is a command given to the computer that instructs the computer to take a specific action, such as display to the screen, or collect input. A computer program is made up of a series of statements.

Statement block : In C, we can link two or more statements together. This is called a block of code or a code block. To create a block of code, we surround the statements in the block with opening and closing curly braces. Once this is done, the statements form one logical unit, which may be used anywhere that a single statement may. For example, the general form of blocks of c ode is

```
{ statement_1;
  Statement_2;
  . . . . . .
  Statement_N;}
```

2.2 The Selection statements

A selection statement selects among a set of statements depending on the value of a controlling expression. selection statements make decisions based upon the outcome of some condition. There are three selection statements in C : ***if, else-if*** and ***switch.***

2.3 The "if" statement

The ***if*** statement allows a program to conditionally execute a statement. General form of the ***if*** is shown here:

```
if(expression) statement;
```

If the expression evaluates as true, the statement will be executed. If it does not, the statement is bypassed, and the line of code following the ***if*** is executed. The statement that follows an if is usually referred to as the ***target*** of the if statement.

Commonly, the expression inside the if compares one value with another using a ***relational*** operator. Examples:

```
if(10 > 9) printf("true");
```

However, because the expression in the following statement is false, the if does not execute its target statement.

```
if(5 > 9) printf("this will not print");
```

```
#include <stdio.h>
int main(void){  int num;
                 printf("Enter an integer : "); scanf("%d", &num);
                 if(num < 0) printf("the number is negative");
                 if(num > -1) printf("the number is non-negative");
                 return 0;}
```

The general form of the ***if*** using blocks of code is

```
if(expression) { statement_1;
                 Statement_2;
                 . . . . . .
                 Statement_N;}
```

If the expression evaluates to true, then all the statements in the block of code associated with the ***if*** will be executed. . If it does not, the statement block is bypassed.

NOTE : *Remember, in C,* ***true*** *is any nonzero value-and* ***false*** *is zero. Therefore, it is perfectly valid to have an* ***if*** *statement such as the one shown here:*

```
if(count+1) printf("Not Zero");
```

2.4 The "if-else" statement

General form of the ***if-else*** is shown here:

```
if(expression) statement_1;
else statement_2;
```

If the expression is true, then the target of the ***if*** will execute, and the ***else*** portion will be skipped. However, if the expression is false, then the target of the ***if*** is bypassed, and the target of the ***else*** will execute.

The general form of the ***if-else*** using blocks of code is

```
if(expression) { statement_1;          else(expression) { statement_1;
                 Statement_2;                             Statement_2;
                 . . . . . .                              . . . . . .
                 Statement_N;}                            Statement_N;}
```

If the expression evaluates to true, then all the statements in the block of code associated with the ***if*** will be executed.If the expression is false, then all the statements in the ***else*** block will be executed.

2.5 The "for" loop

The ***for*** loop is used to repeat a statement or block of statements a specified number of times. Its general form for repeating a single statement is shown here.

```
                    for(initialization; conditional-test , increment) statement ;
```

1. The "initialization" section is used to give an initial value to the variable that controls the loop. This variable is usually referred to as the ***loop-control variable***. The initialization section is executed only once, before the loop begins. The "**conditional-test**" portion of the loop tests the loop-control variable against a target value.
2. If the conditional test evaluates true, the loop repeats. If it is false, the loop stops, and program execution picks up with the next line of code that follows the loop.
3. The **conditional test** is performed at the **start or top of the loop** each time the loop is repeated.
4. The "**increment**" portion of the for is executed at the **bottom of the loop**. That is, the increment portion is executed after the statement or block that forms its body has been executed. The purpose of the increment portion is to increase (or decrease) the loop-control value by a certain amount.

The general form of the ***if-else*** using blocks of code is

```
     for(initialization; conditional-test , increment) { statement_1;
                                                         statement_2;
                                                         . . . . . .
                                                         statement_N;}
```

As a simple first example, this program uses a for loop to print the numbers 1 through 10 on the screen.

```
     #include<stdio.h>
     int main(void){  int num;
                      for(num=1; num<11; num=num+1) printf(" %d ", num);
                      printf(" terminating ");
                      return 0;}
```

The program works like this:

1. First, the loop control variable ***nun*** is initialized to ***1***.
2. The expression ***num < 11*** is evaluated. Since it is true, the for loop begins running.
3. After the number is printed, ***num*** is incremented by ***one*** and the conditional test is evaluated again.
4. This process continues until ***num equals 11***. When this happens, the for loop stops, and *terminating* is displayed.

NOTE :

1. The initialization portion of the ***for loop*** is only executed once, when the loop is first entered.
2. The conditional test is performed at the start of each iteration. This means that if the test is false to begin with, the loop will not execute even once. For example, this program only displays "*terminating*" because ***num*** is initialized to ***11***, causing the conditional test to fail.

```
          for(num=11; num<11; num=num+1) printf(" %d ", num);
          printf(" terminating ");
```

[Important *Note is using* ***num==0*** *instead of* ***num=0*** *cause different result, "==" is a logical operator***]**

3. A ***for*** loop can run negatively. For example, this fragment decrements the loop-control variable.

```
          for (num=20; num>0; num=num-1) . . .
```

4. Further, the loop-control variable may be incremented or decremented by more than one. For example, this program counts to 100 by Fives:

```
          for(i = 0; i<101; i = i+5) printf("%d",i);
```

5. ***Infinite Loop :*** for(num=1; num>0; num=num+1) printf(" %d ", num); or, for(num=1; num<=11||num>11; num=num+1) printf(" %d ", num);

2.6 The increment operator "+ +" and decrement operator "- - "

Increment operator : Recall `for(num=11; num<11; num=num+1)...` , here **num=num+1** can be replaced by **num++** . Although **num=num+1** not incorrect, but never be used in professionally written C programs. C provides a special operator that *increments a variable by one*. The ***increment operator*** is ***"++"*** (two pluses with no intervening space). Using the increment operator, we can change this line of code: **`i = i + 1;`** into this: **`i++;`**

Therefore, the ***for*** will normally be written like this : `for(num=11; num<11; num=num+1). . .`

Decrement operator : Similarly, to decrease a variable by one, we can use C's decrement operator **"--"** (There must be no space between the two minus signs). Therefore, **`count = count - 1;`** can be rewritten as : **`count--;`**

[The reason to use the increment and decrement operators is that, for most C compilers, they will be faster than the equivalent assignment statements.]

The increment and decrement operators do not need to follow the variable [i.e. after variable : i++] ; they can precede it [i.e. before variable : ++i].

Difference between "i++" and "++i" : Although the effect on the variable is the same, the position of the operator does affect when the operation is performed. To see how, examine this program:

```
#include<stdio.h>
     int main(void){ int i,j ;i=10; j=i++;printf("i = %d  and  j = %d ", i,j);return 0;}
     /* this will print i=11 and j=10 */
#include<stdio.h>
     int main(void){ int i,j ;i=10; j=++i;printf("i = %d  and  j = %d ", i,j);return 0;}
     /* this will print i=11 and j=11 */
```

j = i++ works like this : First, the current value of ***i*** is assigned to ***j***. Then ***i*** is incremented. This is why ***j*** has the value ***10***, not ***11***. When the increment or decrement operator follows the variable, the operation is performed after its value has been obtained for use in the expression.
j = ++i works like this : First, the current value of ***i*** is incremented then assigned to ***j***. If the variable is preceded by the increment or decrement operator, the operation is performed first, and then the value of the variable is obtained for use in the expression.

NOTE :

1. If you are simply using the increment or decrement operators to replace equivalent assignment statements, it doesn't matter if the operator precedes or follows the variable. This is a matter of your own personal style.

```
2. int i; i=0;printf(" %d  %d  %d  %d", --i, --i, --i, --i); Gives : -4   -4   -4   -4
3. int i; i=0;printf(" %d  %d  %d  %d", i--, i--, i--, i--); Gives : -3   -2   -1   0
4. int i; i=0;printf(" %d  %d  %d  %d", ++i, ++i, ++i, ++i); Gives : 4   4   4   4
5. int i; i=0;printf(" %d  %d  %d  %d", i++, i++, i++, i++); Gives : 3   2   1   0
```

2.7 The backslash charecters \n "newline", \t "tab" etc

The characters can not be entered from keyboard

\b	Backspace	\0	null
\f	From feed	\\	backslash
\n	Newline	\v	vertical tab
\r	Carriage return (related to ***ENTER*** key)	\a	alert
\t	horizontal tab	\?	Question mark
\"	single quote	\N	Octal constant (N is octal value)
\'	double quote	\xN	hexdecimal constant (N is hexadecimal value)

To print the percentage sign '%' In C via `printf` ***function*** : printf("%%"); /*prints %*/. And printf("10%%"); /*prints 10%*/.
The backslash codes are character constants. To assign one to a character variable, we must enclose the backslash code within a single quote , as shown in the fragment

```
char ch; ch='\t'; /*assign ch the tab charecter*/
```

The ***\n*** newline character does not have to go at the end of the string that is being output by ***printf( )***; it can go anywhere in the string. The point is that there is no connection between a newline and the end of a string. For example, this program:

```
#include <stdio.h>
int main(void){printf("one\ntwo\nthree\nfour\a\a"); return 0;} /*with alert*/
```

2.8 RELATIONAL AND LOGICAL OPERATORS

The ***relational operators*** compare two values and return a true or false result based upon that comparison. The ***logical operators*** connect together true/false results.

Relational operators	
Operator	***Action***
>	Greater than
>=	Greater than or equal
<	Less than
<=	Less than or equal
==	Equal
!=	Not equal
Logical operators	
&&	AND
||	OR
!	NOT

Truth table of logical operators				
p	***q***	***P&&q***	***p||q***	***!p***
0	0	0	0	1
0	1	0	1	1
1	1	1	1	0
1	0	0	1	0

Precedence of the relational and logical operators	
Highest	!
	> >= < <=
	== =!
	&&
Lowest	||

1. An expression like : `10 + count > a * 12` is evaluated as if it were written `(10 + count) > (a + 12)`
2. You may link any number of relational operations together using logical operators. For example, this expression joins three relational operations.
```
var > max || !(max==100) && 0 <= item
```
3. though C defines true as any nonzero value, the relational and logical operators always produce the value 1 for true.
4. You can use the relational and logical operators in both the if and for statements. For example, the following statement reports when both a and b are positive:
```
if(a>0 && b>0) printf("Both are positive.");
```
5. In professionally written C code, it is uncommon to find a statement like this:
```
if(count != 0) . . .
```
the preceding statement is generally written as this:
`if(count)...` The reason is that in C, ***true is any nonzero*** value and ***false is zero***.
6. Statements like this: `if(count == 0)...` are generally written as: `if(!count)...` The expression **`!count`** is true only if **`count`** is zero.
7. It is important to remember that the ***outcome of a relational or logical operation is 0 when false and 1 when true***.
8. C does not define an ***exclusive-OR (XOR)*** logical operator. However, it is easy to create a function that performs the operation.
```
int xor(int a, int b) { return (a || b) && !(a && b) ; }
```
That is, the ***XOR*** operation produces a true result when one and only one operand is true.

9. These two expressions `0&&1||1` and `0&&(1||1)` do not evaluate to the same outcome according to precedence of the logical operators.

10. ***If a program runs for*** `(a==x)||(b==y)||(c==z)||(d==t)` then ***it not runs for*** `(a!=x)&&(b!=y)&&(c!=z)&&(d!=t)`

NOTE : Execution of printf with ++ operators

Consider below C++ program and predict its output.

```
printf("%d %d %d", i, ++i, i++);
```

The above invokes undefined behaviour by referencing both 'i' and 'i++' in the argument list. It is not defined in which order the arguments are evaluated. Different compilers may choose different orders. A single compiler can also choose different orders at different times.For example below three printf statements may also cause undefined behavior.

```
// All three printf() statements in this cause undefined behavior

#include <stdio.h>
int main() {
    volatile int a = 10; printf("%d %d", a, a++);
    a = 10; printf("%d %d", a++, a);
    a = 10; printf("%d %d %d ", a, a++, ++a);
    return 0;}
```

Therefore, it is not recommended ***Not to do two or more than two pre or post increment operators in the same statement***. This means that there's absolutely no temporal ordering in this process. The arguments can be evaluated in any order, and the process of their evaluation can be intertwined in any way.

2.9 Character inputs from keyboard

C defines a function called ***getchar( )***, which ***returns a single character*** typed on the keyboard. When called, the function waits for a key to be pressed. Then ***getchar( )*** echoes the keystroke to the screen and returns the value of the key to the caller.

```
#include <stdio.h>
int main(void){ char ch;
                ch = getchar(); /* read a char :: Similar to scanf * /
                printf(" you typed: %c", ch);
                return 0;}
```

NOTES :

✓ In many C compilers, ***getchar( )*** is implemented in such a way that it line buffers input. That is, it does not immediately return as soon as you have pressed a key, but waits until you have entered an entire line, which may include several other characters. This means that even though it will read and return only one character,

✓ ***getchar( )*** waits until you enter a ***carriage return (i.e., press ENTER)*** before doing so. When ***getchar( )*** returns, it' will return the first character you typed. However, any other characters that you entered, including the carriage return, will still be in the input buffer. These characters will be consumed by subsequent input requests, such as through calls to ***scanf( )***. In some circumstances, this can lead to trouble.

✓ When ***getchar( )*** is implemented in a line-buffered fashion in a modern interactive environment, its use is severely limited.

✓ The function ***getche( )*** which is not defined by the ANSI C standard it can be used like ***getchar( ),*** except that it will return its value immediately after a key is pressed; it ***does not line-buffer input***. for most compilers, this function requires a header file called ***CONIO.H***.

✓ characters returned by either ***getchar( )*** or ***getche( )*** will be represented by their ***ASCII codes***. ASCII character codes are an ordered sequence; each letter's code is one greater than the previous letter, each digit'S code is one greater than the previous digit. This means that ***'a'*** is less than ***'b'***, ***'2'*** is less than ***'3'***, and so on. You may compare characters just like you compare numbers. Example:

```
ch = getchar() ;
if(ch < 'f') printf("character is less than f");
printf("\nits ASCII code is %d", ch);
```

2.10 NEST *if* statements *and* *if-else-if* LADDER

Nested if : When an if statement is the target of another ***if*** or ***else***, it is said to be nested within the outer ***if***. Here is a simple example of a ***nested if***:

```
if(count > max)    /* outer if */
if (error ) printf ("Error, try again. ");     /* nested if */
```

Here, the ***printf( )*** statement will only execute it ***count*** is greater than ***max*** and if ***error*** is nonzero. Notice how the nested if is indented. A nested ***if*** may also appear inside a block of statements that are the target of the outer ***if***.

An ANSI-standard compiler will allow us to nest ifs at least 15 levels deep. (However, it would be rare to find such a deep nesting.)
if-else-if ladder : ***if-else-if ladder*** is another form of ***nested if***. It is possible to string together several ***ifs*** and ***elses*** into what is sometimes called an ***if-else-if ladder*** or ***if-else-if staircase*** because of its visual appearance. In this situation a nested ***if*** has as its target another ***if***. The general form of the ***if-else-if ladder*** is shown here;

FORM 1 :
```
if(expression) statement;
else
   if(expression) statement;
   else
     if(expression) statement;
       .
       .
       .
       else statement;
```

FORM 2 :
```
if(expression) statement;
else { if(expression) statement;
       else { if(expression) statement;
              else { if(expression)
                             statement;
                          .
                          .
                          .
                     else { if(expression)
             statement;
                            else statement;
                          }
                     . . .
                     }
              }
       }
```

Both ***FORM 1*** and ***FORM 2*** act in the same way. The expressions are calculated from the **top - downward**. As soon as a true condition is found. the statement associated with it is executed, and the rest of the ladder is bypassed. If none of the expressions are true, the final else will be executed. If the final else is not present, no action will take place if all expressions are false . The if-else-if ladder is generally written like this :
```
if(expression) statement;
else if(expression) statement;
else if(expression) statement;
.
.
.
else statement;
```
In this example the else related to the first ***if***
```
if(ch=='S') { /* first if */
printf("Enter a number: "); scanf("%d", &y);
if(y) printf("Its square is %d.", y*y); /* second if */
            }
else printf("Make next selection.");
```
Let's consider another example
```
if(p)
    if (q) printf("a and b are true"); /*Everything changes if {} appears*/
else printf ("To which statement does this else apply?");
```
The question suggested by the second ***printf( )*** is: which ***if*** is associated with the ***else*** ? Fortunately, the answer is quite easy: An ***else always associates with the nearest if in the same block that does not already have an else associated with it***. In this example, the ***else*** is associated with the second ***if***.

Types of Nested if :

General Nested if : Type 1	***General Nested if : Type 2***	***If-else-if ladder***

`if(condition 1){`
`     if(condition 2)`
`     else}`
`else  { if(condition 1)`
`        else}`
	`if(cond. 1){`
	`   if(cond. 2){`
	`       if(cond. 3){`
	`           if(cond. 4) /*4th*/`
	`           else    }/*3rd*/`
	`        else   }/*2nd*/`
	`    else    }/*1st*/`
	`else`
	It is a ***filter type*** of nested if. If cond. 4 is true then all four conditions are true . If cond. 4 is false then associated ***else*** will be executed.
		`if(cond. 1) statement;`
		`else if(cond. 2) statement;`
		`else if(cond. 3) statement;`
		`.`
		`.`
		`.`
		`else statement;`

NOTE:
1. Control statement can be passed through a series of conditional –statement using ***if-else-if ladder*** or can be choose to flow through different series of conditional –statement ***(Type-1)*** or the control can be filtered.
2. ***Data Buffer :*** In computer science, a data buffer (or just buffer) is a region of a physical memory storage used to temporarily store data while it is being moved from one place to another. Typically, the data is stored in a buffer as it is retrieved from an input device (such as a microphone) or just before it is sent to an output device (such as speakers). However, a buffer may be used when moving data between processes within a computer. This is comparable to buffers in telecommunication. Buffers can be implemented in a fixed memory location in hardware—or by using a virtual data buffer in software, pointing at a location in the physical memory. In all cases, the data stored in a data buffer are stored on a physical storage medium. A majority of buffers are implemented in software, which typically use the faster RAM to store temporary data, due to the much faster access time compared with hard disk drives. Buffers are typically used when there is a difference between the rate at which data is received and the rate at which it can be processed, or in the case that these rates are variable, for example in a printer spooler or in online video streaming.
   A buffer often adjusts timing by implementing a queue (or FIFO) algorithm in memory, simultaneously writing data into the queue at one rate and reading it at another rate.

2.11 *for Loop – 'Advanced'*

The ***for*** loop in C is significantly more powerful and flexible than in most other computer languages. The reason that ***for*** is so flexible is that the expressions we called the ***initialization***, ***conditional-test***, and ***increment*** portions of the loop are not limited to these narrow roles. The ***for*** loop places no limits on the types of expressions that occur inside it.

1. We do not have to use the initialization section to initialize a loop-control variable.
2. there does not need to be any loop-control variable because the conditional expression may use some other means of stopping the loop.
3. Finally, the increment portion is technically just an expression that is evaluated each time the loop iterates. It does not have to increment or decrement a variable.
4. one or more of the expressions inside ***for*** may be empty. For example, if the loop-control variable has already been initialized outside the for, there is no need for an initialization expression.

EXAMPLE 1 : This program continues to loop until a q is entered at the keyboard. Instead of testing a loop-control variable, the conditional test in this for checks the value of a character entered by the user.

```
    /*This program continues the loop until 'q' is entered*/
    #include<stdio.h>
    #include<conio.h>
    int main(void){ int i; char ch; ch='a'; /*Give ch an initial value*/
        for(i=0; ch!='q'; i++){ printf("pass : %d\n", i); ch=getche(); } return 0;}
```

Here, the condition that controls the loop has nothing to do with the loop-control variable. The reason ***ch*** is given an initial value is to prevent it from accidentally containing a ***q*** when the program begins.

EXAMPLE 2 : This program asks the user for a value and then counts down to zero from this number. Here, the loop-control variable is initialized by the user outside the loop, so the initialization portion of the loop is empty.

```
    /*This program asks the user for a value and then counts down to zero from this number*/
    #include<stdio.h>
    int main(void){int i;
                    printf("Enter an integer for initialization : "); scanf("%d", &i);
                    for( ; i; i--) printf(" %d ", i);
        return 0;}
```

Here the conditional-test is ***i***, it means the loop will continue until the test is true (i.e non-zero ***i>0***) when ***i*** reach to ***0*** the loop stops because ***0*** means ***false*** in C.

EXAMPLE 3: It is perfectly valid ***for*** the loop-control variable to be altered outside the increment section. For example, the following program manually increments i at the bottom of the loop.

```
            #include<stdio.h>
            int main(void){int i;
                        for(i=0; i<10;){printf(" %d ", i);i++;}
              return 0;}
```

EXAMPLE 4: Using the ***for***, it is possible to create a loop that never stops. This type of loop is usually called an ***infinite loop***. The form of this kind of loop is

```
                    for(  ;  ;  ){. . . . .}
```

EXAMPLE 5: Another variation to for is that its target may be empty. For example, this program simply keeps inputting characters until the user types q.

```
    /*This program continues the loop until 'q' is entered (Ver-2)*/
    #include<stdio.h>
    #include<conio.h>
    int main(void){ char ch;
        for(ch=getche(); ch!='q'; ch=getche()); /*Loop without target*/
        printf("Yo !! Found the Q !!");     return 0;}
```

Notice that the statements assigning ***ch*** a value have been moved into the loop. This means that when the loop starts, ***getche( )*** is called. Then, the value of ***ch*** is tested against ***q*** . Next, conceptually, the nonexistent target of the for is executed, and the call to ***getche( )*** in the ***increment*** portion of the loop is executed. This process repeats until the user enters a q. The reason the target of the for can be empty is because C allows null statements.

2.12 *while Loop*

General form of while loop is : `while(expression) statement;`
Of course, the target of while may also be a block of code. The while loop works by repeating its target as long as the expression is true. When it becomes false, the loop stops.
The value of the expression is checked at the top of the loop. This means that if the expression is false to begin with, the loop will not execute even once.

EXAMPLE 1 : A better way to wait for the letter q to be typed is shown here using ***while***. If you compare it to ***Example 5*** in ***Section 2.11***, you will see how much clearer this version is.

```
                    #include<stdio.h>
                    #include<conio.h>
                    int main(void){char ch;
                                ch=getche();
                                while(ch!='q') ch=getche();
                                printf("\nFound the q");  return 0;}
```

EXAMPLE 2 :The following program is a simple code machine. It translates the characters you type into a coded form by adding 1 to each letter. That is, 'A' becomes 'B': and so forth. The program stops when you press ENTER. (The getche( ) function returns \r when ENTER is pressed.)

```
#include<stdio.h>
#include<conio.h>
int main(void){ char ch;
                printf("Enter your massage :"); ch=getche();
                while(ch!='\r'){printf("%c", ch+1); ch=getche(); } return 0;}
```

2.13 *do Loop*

Cs final loop is do, which has this general form:

```
do {
statements
} while(expression);
```

If only one statement is being repeated, the curly braces are not necessary. Most programmers include them, however, so that they can easily recognize that the ***while*** that ends the do is part of a ***do loop***, not the beginning of a ***while*** loop. The ***do loop*** repeats the statement or statements while the expression is true. It stops when the expression becomes false. The ***do loop*** is unique because-it will always execute the code within the loop at least once, since the expression controlling the loop is tested at the bottom of the loop.

EXAMPLE 1 : The fact that do will always execute the body of its loop at least once makes it perfect for checking menu input. For example, this version of the arithmetic program reprompts the user until a valid response is entered.

```
       #include<stdio.h>
       /*If we dont use <conio.h> and getche(), 'Enter the first letter'
       in the 'do' loop appears twice if wrong entry is entered*/
int main(void){int a,b; char ch;
              printf("Do you want to :\n ");
              printf("Add, Substruct, Multply, or Devide?\n ");

    /*Force user to enter a valid response */
    do{printf("Enter the first letter : "); ch=getchar();
       } while(ch!='A' && ch!='S' && ch!='M' && ch!='D');

    printf("\nEnter the first number :\n "); scanf("%d", &a);
    printf("Enter the second number :\n "); scanf("%d", &b);

    if(ch=='A')  printf("Addition of the numbers : %d \n ", a+b);
    else if(ch=='S') printf("Subtraction of the numbers : %d \n ", a-b);
    else if(ch=='M') printf("Multiplication of the numbers : %d \n ", a*b);
    else if((ch=='D') && (b!=0)) printf("Division of the numbers : %d \n ", a/b);
    return 0;}
```

EXAMPLE 2 : The ***do loop*** is especially useful when your program is waiting for some event to occur. For example, this program waits for the user to type a ***q*** . Notice that it contains one less call to ***getche( )*** than the equivalent program (example 1) described in 2.12 on the while loop.

```
#include<stdio.h>
#include<conio.h>
int main(void){char ch;
               do {ch=getche();} while(ch!='q');
               printf("\nFound the q");
               return 0;}
```

2.14 NESTED LOOPS

When the body of one loop contains another, the second is said to be nested inside the first. Any of C's loops may be nested within any other loop.
The ANSI C standard specifies that loops may be nested at least ***15 levels deep***. However, most compilers allow nesting to virtually any level.
As a simple example of nested fors, this fragment prints the numbers 1 to 10 on the screen ten times.

```
for(i=0; i<10; i++) {
for(j=l; j<11; j++) printf ("%d ", j); /* nested loop */
printf("\n“);}
```

Let us see another example

```
#include<stdio.h>
int main(void){
    int i, ans, crct, right;
    for(i=1; i<11; i++){
      printf("\nWhat is %d + %d = ",i,i); scanf("%d", &ans);
      if(ans==(i+i)) printf("Correct");
      else {printf("False");
            printf("\nTry again");
            right=0;       /*When ans is false right=0 so that !right=1 equivalent to false=1*/
```

```
                /*Nested for*/
        for(crct=1; (crct<3)&&(!right); crct++ ){
            printf("\nWhat is %d + %d = ",i,i); scanf("%d", &ans);
                if(ans==(i+i)){ printf("Correct"); right=1;}   /*right changes its value*/
                else {printf("False"); printf("\nTry again");}
                }
                /*if answer is still wrong tell user*/
        if(right==0) printf("\nFalse. The correct ans is %d + %d= %d",i,i,(i+i));
        }
    }
    return 0;}
```

The ***for loop*** continues only if the conditional test is true (i.e. 1), hence we set ***right=0*** so that ***(crct<3)&&(!right)*** act as ***1 && 1 = 1*** (see ***truth table for logical statement***). In this case either ***(crct<3)*** or ***(!right)*** is '***zero***' the loop stops. The reason to set ***right=0*** is that in the ***(crct<3)&&(!right)*** statement ***(!right)=1*** so that the loop continues if the ***answer is false***.

2.15 Use **break** TO EXIT A LOOP

The break statement allows you to exit a loop from any point within its body, bypassing its normal termination expression. When the break statement is encountered inside a loop, the loop is immediately stopped, and program control resumes at the next statement following the loop.
For example, this loop prints only the numbers 1 to 10:

```
            for(j=l; j<100; j++) {printf ("%d ", j);
                                  if(j==10) break; /*exit the loop*/
                                  }
```

NOTE :

1. The break statement can be used with all three of C's loops.
2. You can have as many break statements within a loop as you desire.
3. Since too many exit points from a loop tend to destructure your code, it is generally best to use the break for special purposes, not as your normal loop exit.
4. The reason C includes the break statement is to allow your programs to be more efficient. For example, examine this fragment:

```
            do { printf("Load, Save, Edit, Quit?\n");
                  do {printf("Enter your selection:"); ch = getchar();
                     } while(ch!='L' && ch!='S' && ch!='E' && ch!='Q');
                  if(ch != 'Q') { /*do something */ }
                  if(ch != 'Q') {/*do something else*/}
                               /* etc. */
            } while(ch != 'Q');
```

In this situation, several additional tests are performed on ch to see if it is equal to 'Q' ***to avoid executing certain sections of code when the Quit option is selected (which makes the program act slow)***. Most C programmers would write the preceding loop as shown here:

```
            for( ; ; ) { /* infinite for loop */
                          printf("Load, Save, Edit, Quit?\n");
                        do {printf("Enter your selection:"); ch = getchar();
                           } while(ch!='L' && ch!='S' && ch!='E' && ch!='Q');
            if (ch == 'Q') break;        /*Quit makes exit the infinite for loop */
            /* do something */
            /*do something else*/
            /* etc. */
            }
```

In this version, ***ch*** need only be tested once to see if it contains a ***'Q'***. This implementation is more efficient because **only one if** statement is required.

2.16 Use **continue** for skipping any code

The continue statement is somewhat the opposite of the break statement. It forces the next iteration of the loop to take place, skipping any code in between itself and the test condition of the loop. For example, this program never displays any output:

```
  #include<stdio.h>
  int main(void){ int x;
                for(x=0; x<100; x++){continue;
                    printf("this never show up"); /* this is never executed */ }
                return 0;}
```

Each time the continue statement is reached, it causes the loop to repeat, skipping the printf( ) statement.

NOTE:

1. In ***while*** and ***do-while*** loops, a ***continue*** statement will cause control to go directly to the ***test condition*** and then continue the looping process.
2. In the case of ***for***, the increment part of the loop is performed, the conditional test is executed, and the loop continues.

2.17 The **switch** STATEMENT

The switch statement is C's multiple selection statement. It is used to select one of several alternative paths in program execution and works as follows :
A value is successively tested against a list of integer or character constants. When a match is found. the statement sequence associated with that match is executed. The general form of the switch statement is this:

```
switch(value) {     case constant_1:statement sequence; break;
                    case constant_2: statement sequence; break;
                    case constant_3: statement sequence: break;
                    . . .
                     . . .
                    default : statement sequence; break;}
```

The default statement sequence is performed if no matches are found. The default is optional. If all matches fail and default is absent, no action takes place. When a match is found, ***the statements associated with that case are executed until break is encountered or, in the case of default or the last case, the end of the switch is reached.***

As a very simple example, this program recognizes the numbers 1, 2. 3, and 4 and prints the name of the one you enter. That is, if youenter 2, the program displays two.

```
#include<stdio.h>
int main(void){int x;
               printf("Enter a number between 1 and 4 :"); scanf("%d", &x);
switch(x){   case 1 : printf("One"); break;
             case 2 : printf("Two"); break;
             case 3 : printf("three"); break;
             case 4 : printf("four"); break;
             case 5 : printf("five"); /*In absence of 'break' control jumps to Next case (case 6)*/
             case 6 : printf("fuck !!"); break;
             default : printf("Not acceptable 5 and more");break;}
    return 0;}
```

Nested switch : It is possible to have a switch as part of the statement sequence of an outer switch. This is called a nested switch. If the case constants of the inner and outer switch contain common values, no conflicts will arise. For example, the following code fragment is petfectly acceptable:

```
     switch(a) { case 1:
                    switch(b) {  case 0: printf("b is false"); break;
                                 case 1: printfC"b is true"); break; /*'break' may be absent*/
                               }break ;
                  case 2:. . . . . .
```

NOTE :

1. The ***switch*** statement differs from ***if*** in that ***switch*** can only test for equality, whereas the ***if*** conditional expression can be of any type.
2. ***switch*** will work with only ***int*** or ***char*** types. You cannot, for example, use ***floating-point numbers***.
3. The statement sequences associated with each case are ***not blocks***; they are not enclosed by curly braces.
4. The ***ANSI C*** standard states that at least 257 case statements will be allowed. In practice, you should usually limit the amount of case statements to a much smaller number for efficiency reasons.
5. Also ***no two case constants in the same switch can have identical values***.
6. An ANSI-standard compiler will allow at least 15 levels of nesting for switch statements.
7. Technically, the ***break*** statement is optional. The ***break*** statement, when encountered within a ***switch***, causes the program flow to exit from the entire ***switch*** statement and continue on to the next statement outside the ***switch***. This is much the way it works when breaking out of a loop. However, ***if a break statement is omitted, execution continues into the following case or default statement*** (if either exists). **When a break statement is missing, execution "falls through" into the next case and stops only when a break statement or the end of the switch is encountered.**
8. The statement sequence associated with a case may be empty. This allows two or more cases to share a common statement sequence without duplication of code.

2.18 The **goto** STATEMENT

C supports a ***non-conditional jump*** statement, called the ***goto***. Because C is a replacement for assembly code, the inclusion of ***goto*** is necessary because it can be used to create very fast routines. However, most programmers do not use ***goto*** because it destructures a program and, if frequently used, can render the program virtually impossible to understand later. Also, there is no routine that requires a ***goto***.
The ***goto*** statement can perform a ***jump*** within a function. It cannot jump between functions. It works with a label. In C, a label is a valid identifier name followed by a colon. For example, the following ***goto*** jumps around the ***printf( )*** statement:

```
goto mylabel;
printf("This will not print ");
mylabel: printf ("This will print. ") ;
```

About the only good use for goto is to jump out of a deeply nested routine when a catastrophic error occurs.

2.19 C control statements

Any c program can be constructed from only seven different types of control statements (sequence, if, if...else, switch, while, do...while and for) combined in only two ways (control- statement stacking and control-statement nesting).

C provides three types of ***selection statements*** (discussed in this chapter). The ***if*** selection statement is a **single-selection statement** because it selects or ignores a single action (or, as we'll soon see, a single group of actions). The ***if...else*** statement is called **a double-selection statement** because it selects between two different actions (or groups of actions). The ***switch*** selection statement is called a **multiple-selection statement** because it selects among many different actions (or groups of actions).

C provides three types of ***repetition statements*** that enable programs to perform statements repeatedly as long as a condition remains true. The repetition statements are the ***while***, ***do...while*** and ***for*** statements. The ***while*** and ***for*** statements perform the action (or group of actions)in their bodies zero or more times—if the loop-continuation condition is initially false, the action (or group of actions) will not execute. The ***do...while*** statement performs the action (or group of actions) in its body at least once.

Another classification is :

1. Labeled Statements
2. Compound Statements
3. Expression Statements
4. Selection Statements
5. Iteration Statements
6. jump Statements

Labeled Statements : A statement can be preceded by a label. Three types of labels exist in C. A simple identifier followed by a colon (:) is a ***label***. Usually, this label is the target of a ***goto*** statement. Within ***switch*** statements, ***case*** and ***default labeled*** statements exist.
1) `label : statement ; ...  goto label;` ***2)*** `case int/char-const : statement;` ***3)*** `default : statement;`

A compound statement (statement block) : A compound statement is the way C groups multiple statements into a single statement. It consists of multiple statements and declarations within braces (i.e. { and }).

Expression Statements: An expression statement consists of an optional expression followed by a semicolon (;). If the expression is present the statement may have a value. If no expression is present the statement often called the null statement. The printf function calls are expressions, so statements such as printf ("Hello World\n"); are expression statements.

Selection Statements : There are three selection statements in C : ***if, else-if*** and ***switch.***

Iteration Statements : There are three iteration statements in C : ***while, do....while, for.***

jump Statements : C has four types of jump statements.

1. ***goto statement :*** the form is `goto identifier;`
   This statement transfers control flow to the statement labeled with the given identifier. The statement must be within the same function as the goto.
2. ***break statement*** : the form is `break;`
   is used within ***iteration statements*** and ***switch*** statements to pass control flow to the statement following the ***while***, ***do-while***, ***for***, or ***switch***.
3. ***continue statement :***the form is `continue;`
   is used within the sub-statement of iteration statements to transfer control flow to the place just before the end of the sub-statement.
4. ***return statement :*** the form is return expression$_{(opt)}$;
   This statement returns from the function. If the function return type is void, the function may not return a value; otherwise, the expression represents the value to be returned.