1.7 DECISION MAKING, BRANCHING and LOOPING

1.7.1 Introduction

A set of statements that alter the flow of control of program execution fall under the category of **control-flow statements**. For most of the problems the design of programs will need control flow statements. A set of statements may have to be executed based upon certain conditions. For instance, you may want to avoid finding factorial of a number when a given number is negative. In this case a decision is to be taken after checking the condition whether the number is negative or positive. If it is negative print an error message, else simply go on to find its factorial.

C Language provides `if`, `if-else`, `switch`, `goto`, `continue` and `break` statements to handle decision making. You will use logical and relational operators to test for certain conditions.

1.7.2 The if Statement

The formal syntax of if statement is,

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

Table 1.9 Mathematical Functions

Sl. No.	Name of the function	Meaning
1	sin(x)	sine
2	cos(x)	cosine of x
3	tan(x)	tangent of x
4	asin(x)	$\sin^{-1}(x)$
5	acos(x)	$\cos^{-1}(x)$
6	atan(x)	$\tan^{-1}(x)$
7	atan2(y,x)	$\tan^{-1}(y/x)$
8	sinh(x)	hyperbolic sine of x
9	cosh(x)	hyperbolic cosine of x
10	tanh(x)	hyperbolic tangent of x
11	exp(x)	e^x
12	log(x)	ln(x) natural log, x>0
13	log10(x)	$\log_{10}(x)$, x>0
14	pow(x,y)	x^y
15	sqrt(x)	sqrt of x
16	ceil(x)	smallest integer not less than x, as a double
17	floor(x)	largest integer not greater than x, as a double
18	fabs(x)	absolute value of x, |x|
19	ldexp(x,n)	x. 2^n
20	fmod(x,y)	floating point remainder of x/y

Here, `condition` is an expression involving relational and logical operators. The `statement` may be a single legal C statement or a compound statement enclosed within braces (see below)

```
if (condition)
{
      statement-1;
      statement-2;
            :
            :
      statement-n;
}
```

Execution Sequence

When the program control comes to the `if` statement, the condition is evaluated first. If the condition is evaluated as *true* (any positive value is taken as true), the control goes to statement(s) following the `if` statement. Supposing, if the condition is *false* (zero), then the statements inside the curly braces are skipped.

Example program 1.14
To find the bigger out of two numbers.

```
#include <stdio.h>
#include <stdlib.h>
void main()
{
     int num1;  /* first data */
     int num2;  /* second data */
     printf("Enter the first number:");
     scanf("%d",&num1);
     printf("Enter the second number:");
     scanf("%d",&num2);
     if (num1 > num2)
     {
          printf("The  bigger one is = %d\n",num1);
          exit(0);
     }
     printf("the bigger one is = %d\n",num2);
}
```

Sample Run-1

```
Enter the first number:7
Enter the second number:5
The bigger one is = 7
```

Sample Run-2

```
Enter the first number:5
Enter the second number:28
The bigger one is = 28
```

In the first case, i.e., num1 = 7 and num2 = 5, num1 is greater than num2 and therefore, `if` statement expression evaluates to *true*. The bigger one (num1) is printed and the program exits, because of `exit(0)` function. The `exit()` function takes the control back to the operating system unconditionally.

In the second case, since the first number is not greater than the second, no statement under `if` is executed. The control comes to the last statement i.e., `printf()` function and prints the second number as bigger.

Program 1.15
if statement – more closer look

```
#include <stdio.h>
void main()
{
     int a;
     printf("Enter an integer:");
     scanf("%d",&a);
     if (a=5)
       printf("entered  value is = %d/n",a); /* line-1 */
}
```

Sample Run-1

```
Enter an in integer:5
Entered value is = 5
```

Sample Run-2

```
Enter an integer:7
Entered value is = 5
```

Well something is wrong here! Just look at the second output. It has given a wrong result why? The mistake is in statement `if (a = 5)`. The condition `a = 5` is not really a conditional statement, instead it is an assignment statement. The compiler does not give any error. It actually evaluates to *true* (or 1). In other words, whatever be the value of a, it always executes `printf()` in line-1

You must write:

```
if (a == 5)
        printf ("---- ");
```

```
            case constant2: statement(s);
                            break;

            .....

                    default: statement(s);
                             break ;    /* optional */
}
next-statement; /* line-1 */
```

Execution sequence

- The switch expression is evaluated first.
- Each case is labeled by one or more integer-valued constants. If a `case` matches the expression value, execution starts at the `case` until it encounters `break`. After executing `break`, the control comes to line-1 (i.e. next-statement).
- The `case` labeled the `default` is executed if none of the other cases are satisfied.
- The `default` keyword is optional.
- The `break` under the `default` case is also optional.
- In case there is no `break` for a particular `case` structure; the execution proceeds with the next `case`. In other words, the control does not come to line-1.
- There must be at least one blank space between the keyword `case` and the `constant` as per the syntax rule.

Program 1.17
Simple function calculator

```
#include <stdio.h>
void main()
{
    float opnd1;          /* first operand      */
    float opnd2;          /* second operand     */
    char op;              /* operator           */
    float result = 0;     /* final answer       */

    printf(" enter an expression :");
    scanf("%f%c%f", &opnd1, &opnd2);
    /* Evaluate */
    switch (op)
    {
        case '+': result = opnd1+ opnd2;
                  break;
        case '-': result = opnd1 - opnd2;
                  break;
        case '*': result = opnd1 * opnd2;
```

```
                    break;
            case '/': result= opnd1 / opnd2;
                    break;
            default:  printf("Bad operator \n");
                    break; /* exit(0); */
        }
        /* print the answer */
        printf("%.1f %c %.1f = %.1f/n",opnd1,op,opnd2,result);
}
```

Sample Run

```
Enter an expression:3.2+7
3.2 + 7.0 = 10.2
```

1.7.6 The ? : Operator (Conditional Operator)

The conditional operator is an alternative way to use `if-else` construct.

```
if (conditional expression)
     expression1;
else
     expression2;
```

Now with the C language's short-hand notation (conditional operator ? :), you can accomplish the above task as shown below:

```
conditional expression ? expression1 : expression2;
```

The conditional operator is a *ternary* operator, which means that is has three operands. First operand is the control that precedes the question mark ? Second is `expression1` that precedes colon (:), and the third is `expresion2` that follows the colon(:).

Program 1.18
Finding Max

```
#include <stdio.h>
void main( )
{
     int max;
     int num1;
     int num2;
     printf("Enter two numbers:");
     scanf("%d%d",&num1,&num2);

     max = (num1>num2) ? num1 : num2; /* line-1*/
```

Sample Run

```
4
3     when n = 0 the value of n is printed and
2     the loop exits.
1
0
```

1.7.9 The do-while Loop

To achieve looping, another possibility is to use `do-while` and it is same as while statement except that the testing is done at the bottom.

```
(1) do
        statement
    while (condition);
(2) do
    {
        statement(s);
    } while(condition);
```

Execution Sequence

The statement(s) are executed at least once and then the conditional expression is evaluated. If it turns out to be *true*, the loop repeats, else if the condition is *false* the control goes to the statement following the `do-while`. The semicolon (;) after the `while` statement is a must.

Normally, `do-while` is used when certain statements are executed at least once as it is shown below:

```
do
{
     printf ("Enter an integer number:");
     scanf("%d", &i);
} while (i > 0);
```

The number is read first and if it is positive the loop is executed again-reads another number. In case if the number is negative, the condition is *false* and reading operation is broken, this piece of code does the job of reading only positive values.

1.7.10 The *for* Statement

The `while` and `do-while` are called as **indefinite loop** constructs (because the number of iterations are unknown), where as `for` statement is a definite loop.

```
for (initialize; test; update)
     statement;
```

```
for (initialize; test; update)
{
    statement(s) ;    /*  body of the loop */
}
```

Here, `initialize`, `test`, `update` are optional, but semicolon (;) is a must.

Execution Sequence

The initialize part is executed first and only once. The expression under `test` is evaluated. If it is *true* then the body of the loop is executed. The expression given in the `update` is executed and again it is tested. If it is *true* the iteration continues, otherwise the loop terminates.

Program 1.21
To find b^n raise base to n-th power, n >= 0

```
#include <stdio.h>
void main()
{
int base; /* base  */
    int i, n;  /* power */
    int result;      /* final result */
    printf("Enter base and n:");
    scanf("%d%d", &base, &n);
    result = 1;
    if (n >= 0)
    {
        for (i = 1; i <= n; i++)
            result = result * base;
        printf("%d power %d is = %d\n", base,n,result);
    }
}
```

Sample Run

```
Enter base and n: 2 5
2 Power 5 is = 32
```

It is possible to define multiple variables inside a `for` loop using comma (,) operator. For example,

```
void main()
{
    int cu, cd;              /* countup and countdown */
    for (cu = 0, cd = 10; cu < 10, cu++, cd--)
        printf ("%d\t%d\n", cu,cd);
}
```

Program 1.23
To print the numbers from 1 to 91 in the following fashion, using nested for loop.

```
 1
 2   3
 4   5   6
 7   8   9  10
11  12  13  14  15
16  17  18  19  20  21
22  23  24  25  26  27  28
29  30  31  32  33  34  35  36
37  38  39  40  41  42  43  44  45
46  47  48  49  50  51  52  53  54  55
56  57  58  59  60  61  62  63  64  65  66
67  68  69  70  71  72  73  74  75  76  77  78
79  80  81  82  83  84  85  86  87  88  89  90  91
```

```
#include <stdio.h>
void main()
{
    int i,j;          /* loop indices  */
    int count = 1;    /* to keep track of the sequence */
    /* outer loop controls the rows */
    for (i = 1; count <= 91; i++)
    {
        /* inner loop prints the numbers */
        for (j = 1; j <= i; j++)
        {
              printf("%4d",count);
              count++;
        }
        /* print a blank line after each row */
        printf("\n");
  }
}
```

Program 1.24
Binary to decimal conversion

```
#include <stdio.h>
void main()
{
    int n;          /* binary input                    */
    int a    = 0;   /* to hold the decimal vlaue       */
    int t    = 1;   /* to keep track of powers of two */
```

```
    int flag = 0;     /* flag to catch wrong input        */
    long num;         /* temporary variable               */
    printf("Enter the Binary Number : ");
    scanf("%ld", &n);
    while (n > 0 && !flag)
    {
          num = n % 10;
          if (num == 0 || num == 1)
          {
                a = a+ num * t;
                n = n/10;
                t *= 2;
          }
          else flag++;
    }
    if (!flag) printf("The Decimal Number = %d\n",a);
    else printf("### Wrong Input\n");
}
```

1.8 ARRAYS

1.8.1 Introduction

So far we have seen data elements which are *scalars*. This means that any variable you declare can hold a single value only. However, there are many occasions where in a large amount of data is to be manipulated. In such case cases, it may require thousands of such declarations. For instance, to handle the marks of 80 students in a class- it is cumbersome to declare 80 variables.

Instead, if we can declare a single variable and store the marks for 80 students, we would get the most flexible way of programming. C language provides a facility to solve this problem in the form of an array declaration. An **array** is one that allows you to define large amount of storage for related data elements with a single name.

```
float stdmrk[80];
```

Now, you can handle the marks of the first student with indexing mechanism i.e., `stdmrk[0]`. The following section will give you more details for handling arrays.

1.8.2 One-Dimensional Arrays

Just like any other variable(s), you must declare an array before it could be used. The syntax for the array declaration is:

```
<data_type>  array_name[array_size];
```

For example,

```
int a[10];  /* declares an integer type  array */
float x[5]; /* declares a float array */
int temp[TEN*4]; /* TEN - symbolic constant */
char alpha[26]; /* character array */
```

Syntactic Rule -1

The example (1) declares an integer array, where all the elements must be of integer type. You can't mix up the data types across the defined array type.

Syntactic Rule - 2

To specify the array size, you cannot use a *floa*ting-point number, because compiler cannot allocate fractional memory.

```
int a[7.5];      /* illegal */
```

Whenever the compiler encounters an array declaration, it sets aside a block of memory as specified by the `array_size`.

Program 1.25
Reading and printing

```
#include <stdio.h>
void main()
{
     int a[10]; /* integer array of 10 elements */
     int i;
     printf("Enter 10 integer elements: \n");
     for (i = 0; i < 10; i++)
          scanf("%d",&a[i]); /* read element by element */

     /* print each element in a line */
     for (i = 0;  i < 10; i++)
     {
          printf("%d\n", a[i]);
     }
}
```

Sample Run

```
Enter 10 integer elements:
3 1 5 8 2 7 6 4 9 10
3
1
5
```