CHARUTAR VIDYA MANDAL'S
SEMCOM
Vallabh Vidyanagar

Faculty Name: Ami D. Trivedi
Class: FYBCA
Subject: US01CBCA01 (Fundamentals of Computer Programming Using C)

***UNIT – 3 (Structured Programming, Library Functions and Arrays)**

LOOP CONTROL STRUCTURES

A looping process, in general, would include the following four steps:

1. Setting and initialization of a counter.
2. Execution of the statements in the loop.
3. Test for a specified condition for execution of the loop.
4. Incrementing the counter.

The test may be either to determine whether the loop has been repeated the specified number of times or to determine whether a particular condition has been met.

The C language provides for three loop constructs for performing loop operations. They are:

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

1. While Statement

The basic format of the while statement is

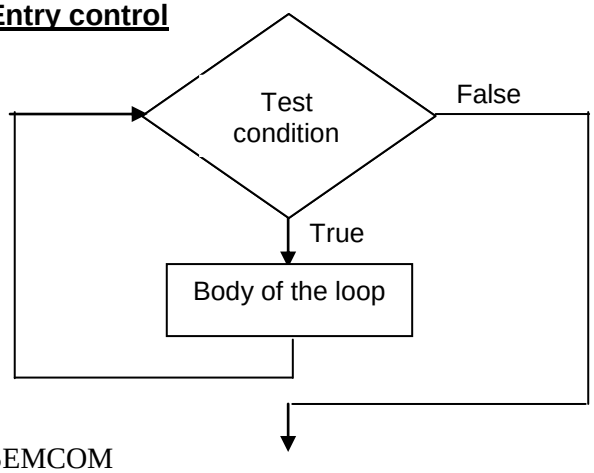
```
while  (test condition)
       {
           body of the loop
       }
```

The while is an entry-controlled loop statement.

- The test-condition is evaluated and if the condition is true, then the body of the loop is executed.
- After execution of the body, the test-condition is once again evaluated and if it is true, the body is executed once again.
- This process of repeated execution of the body continues until the test-condition finally becomes false and the control is transferred out of the loop.
- On exit, the program continues with the statement immediately after the body of the loop.

The body of the may have one or more statements. The braces are needed only if the body contains two or more statements. However, it is a good practice to use braces even if the body has only one statement.

Entry control



e.g.
```
……..
sum = 0;
n = 1;
while(n <= 10)
{
        sum = sum + n;
        n = n+1;
}
printf("sum = %d\n",sum);
……..
```

The body of the loop is executed 10 times for n=1, 2, … 10. Each time value of n will be added to sum variable and n is incremented inside the loop.

This is an example of counter-controlled loops. Variable n is called counter or control variable.

2. Do while statement

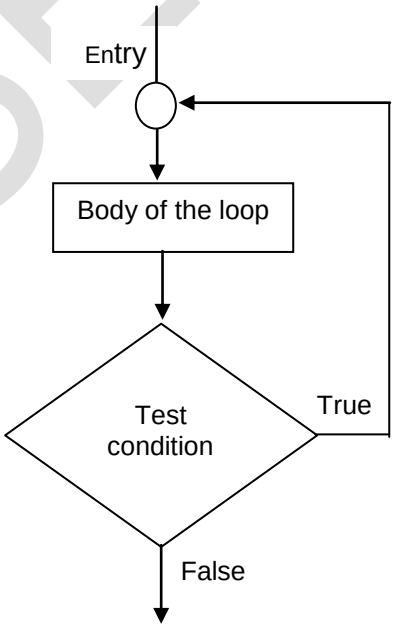
On some occasions it might be necessary to execute the body of the loop before the test is performed. Such situations can be handled with the help of the do statement. This takes the form:

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

The do while is an exit -controlled loop statement.

- On reaching the do statement, the program proceeds to evaluate the body of the loop first.
- At the end of the loop, the test-condition in the while statement is evaluated.
- If the condition is true, the program continues to evaluate the body of the loop once again.
- This process continues as long as the condition is true.
- When the condition becomes false, the loop will be terminated and the control goes to the statement that appears immediately after the while statement.

Exit control



Since the test-condition is evaluated at the bottom of the loop, the do…while construct provides an exit-controlled loop and therefore the body of the loop is always executed at least once.

A simple example of a do….while loop is:

```
……….
do
   {
      printf("Input a number\n");
      scanf("%d",&number);
   }
while (number > 0);
……….
```

This segment reads a number from the keyboard until a zero or a negative number is keyed in.

The do…while construct is useful in data validation.

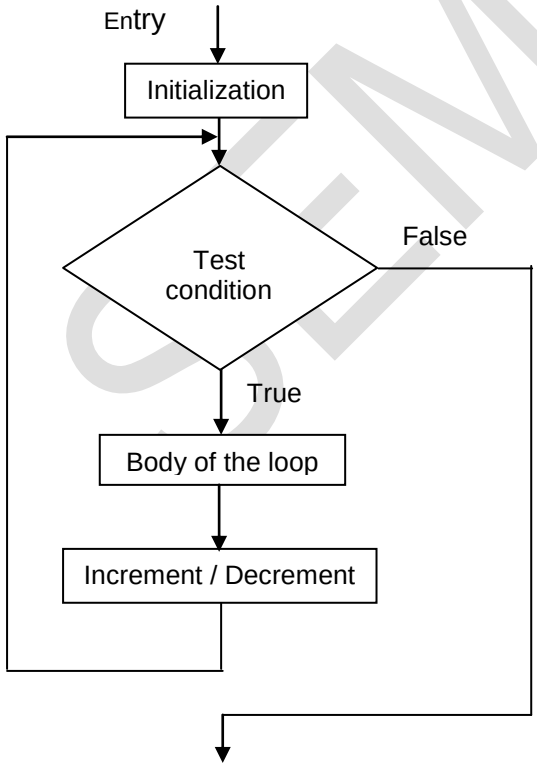
3. For statement

Simple 'for' loop

The for loop is another entry-controlled loop that provides a more concise loop control structure. The general form of the loop is

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

Entry control



The execution of the for statement is as follows:

1. **Initialization** of the control variables is done first. It is done using assignment statements such as i=1 or count=0. the variables I and count are known as loop-control variables.
2. The value of the control variable is tested using the **test-condition**. The test-condition is a relational expression such as i < 10. This expression determines that when the loop will exit.

   If the condition is true, the body of the loop is executed. Otherwise the loop is terminated and the execution continues with the statement that immediately follows the loop.
3. When the body of the loop is executed, the control is transferred back to the for statement after evaluating the last statement in the loop. Now the control variable is incremented / decremented using an assignment statement such as i=i+1.

   And the new value of the control variable is again tested to see whether it satisfies the loop condition. If the condition is satisfied, the body of the loop is again executed. This process continues till the value of the control variable fails to satisfy the test-condition.

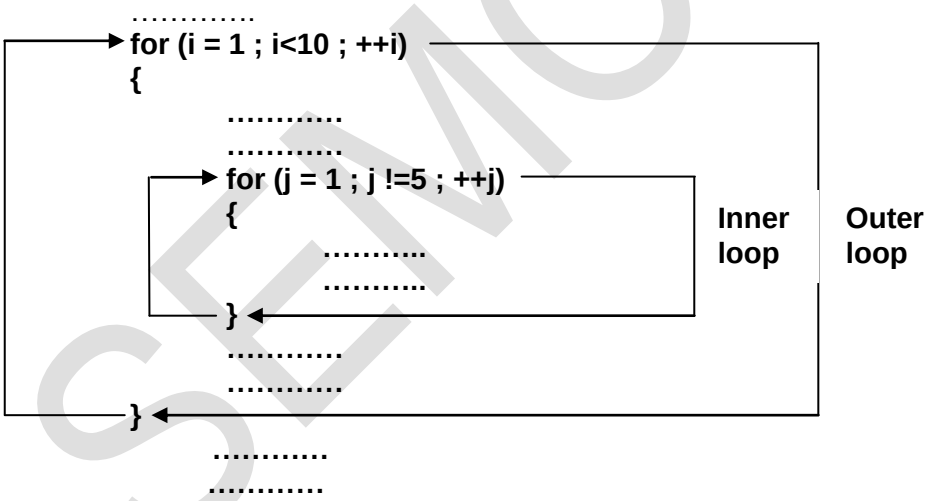
e.g.

```
for (x = 0 ; x <= 9 ; x = x+1)
   {
        printf("%d",x);
   }
```

This for loop is executed 10 times and prints the digits 0 to 9 in one line.

Nesting of for loops

Nesting of loops, that is, one for statement within another for statement is allowed in C. For e.g., two loops can be nested as follows:



The nesting may continue upto 15 levels in ANSI C; many compilers allow more.

Comparison of the three loops

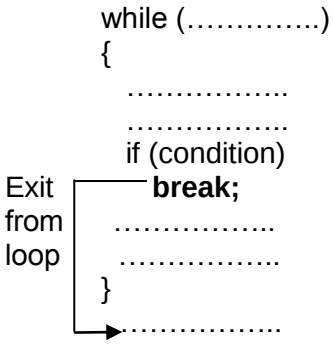
for	while	do

for (n=1 ; n<=10 ;++n)
{
……..
……..
}
	n = 1;
	while(n<=10)
	{
	……
	……
	n = n+1;
	}
		n = 1;
		do
		{
		……
		……
		n = n+1;
		}
		while(n<=10);

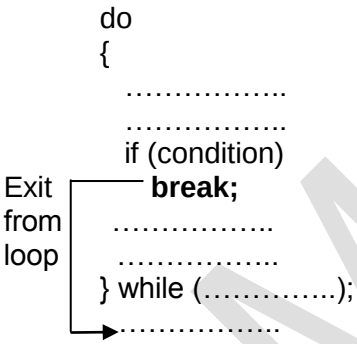
BREAK - Jumping out of a loop

An early exiting from a loop can be accomplished by using the **break** statement or the go to statement. Break can used with if…else, switch, and also within **while, do or for** loops.

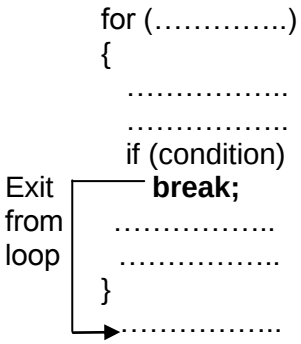
When a break statement is encountered inside a loop, the loop is immediately exited and the program continues with the statement immediately following the loop. When the loops are nested, the break would only exit from the loop containing it. That is, the **break** will exit only a single loop.



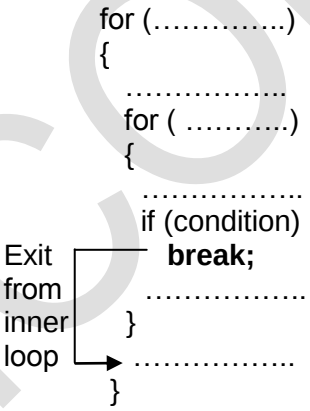
(a)



(b)



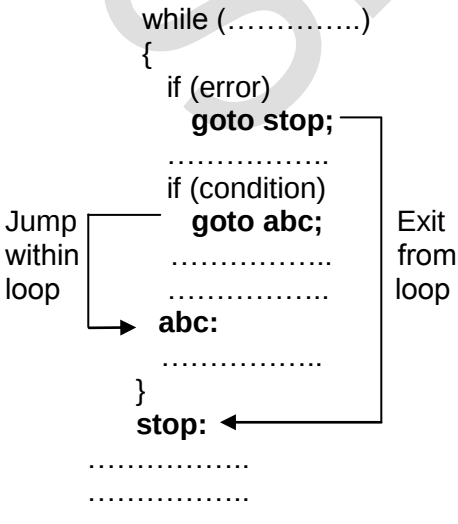
(c)



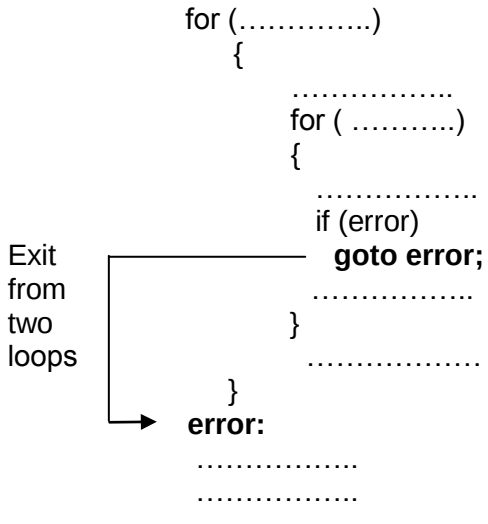
(d)

Since a goto statement can transfer the control to any place in a program, it is useful to provide branching within a loop. Another important use of go to is to exit from deeply nested loops when an error occurs. A simple **break** statement would not work here.

Jumping within and exiting from the loops with goto statement



(a)



(b)

CONTINUE – Skipping a Part of a Loop

During the loop operations, it may be necessary to skip a part of the body of the loop under certain conditions.

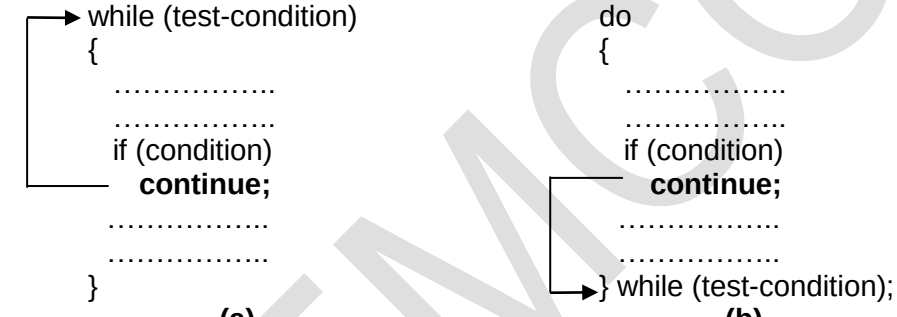
For example, in processing of applications for some job, we might like to exclude the processing of data of applicants belonging to a certain category.

On reading the category code of an applicant, a test is made to see whether his application should be considered or not. If it is not to be considered, the part of the program loop that processes the application details is skipped and the execution continues with the next loop operation.

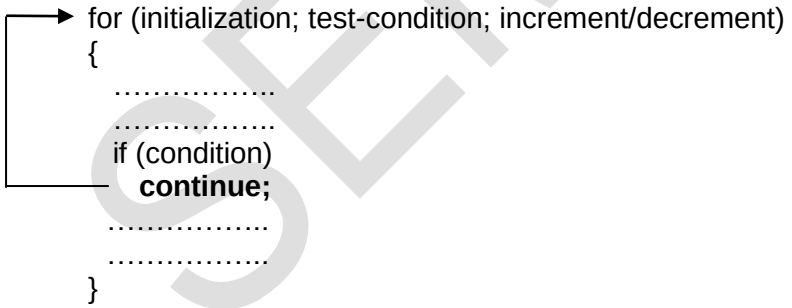
Like the break statement, C supports another similar statement called the **continue** statement. Break statement causes the loop to be terminated. The continue statement causes the loop to be continued with the next iteration after skipping any statements in between.

The continue statement tells the compiler: "SKIP THE FOLLOWING STATEMENTS AND CONTINUE WITH THE NEXT ITERATION". The format of the continue statement is:
continue;

The use of the continue statement in loops is illustrated in following figure. In while and do loops, continue causes the control to go directly to the test-condition and then to continue the iteration process. In the case of for loop, the increment section of the loop is executed before the test-condition is evaluated.



(a) **(b)**



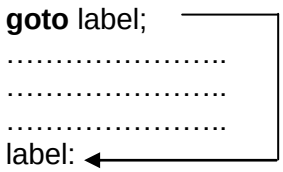
(c)

GOTO STATEMENT

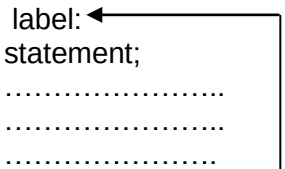
C supports the **goto** statement to branch unconditionally from one point to another in the program. Although it may not be essential to use the **goto** statement in a highly structured language like C, there may be occasions when the use of go to might be desirable.

The **goto** requires a label in order to identify the place where the branch is to be made. A **label** is any valid variable name, and must be followed by a colon.

The label is placed immediately before the statement where the control is to be transferred. The general forms of **goto** and label statements are shown below:



Forward jump



Backward jump

The **label:** can be anywhere in the program either before or after the **goto** label; statement.

During running of a program when a statement like **goto begin;** is met, the flow of control will jump to the statement immediately following the label **begin:**. This happens unconditionally.

Note that a **goto** breaks the normal sequential execution of the program. If the label: is before the statement **goto** label; a loop will be formed and some statements will be executed repeatedly. Such a jump is known as a backward jump.

On the other hand, if the label: is placed after the goto label; some statements will be skipped and the jump is known as a forward jump.

A goto is often used at the end of a program to direct the control to go to the input statement, to read further data. Consider the following example:

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

This program is written to evaluate the square root of a series of numbers read from the terminal. The program uses **goto** statement to skip any further computation when the number is negative.

Note:

It is a good practice to avoid using gotos. There are many reasons for this. When goto is used, many compilers generate a less efficient code. In addition, using many goto makes a program logic complicated and makes the program unreadable.

STRUCTURED PROGRAMMING

Structured programming is an approach to the design and development of programs. It is a discipline of a making a program's logic easy to understand by using only 3 control structures:

1. Sequence structure
2. Selection (branching) structure
3. Repetition (looping) structure

The use of structure programming techniques helps and ensures well designed programs that are easier to write, read, debug and maintain compared to unstructured programs.

Structured programming tries to prevent the implementation of unconditional branching using jump statements such as goto, break and continue. Structured programming means "goto less programming".

* COMMON STANDARD LIBRARY FUNCTIONS

1. <math.h>

abs() and fabs()

Header file: <MATH.H>

abs gives the absolute value of an integer.
fabs gives the absolute value of a floating-point number.

Syntax: abs(int x); fabs(double x);

```
#include <stdio.h>
#include <math.h>
void main()
{
  int number = -1234;
  printf("\nNumber : %d",number)
  printf("\nAbsolute value : %d",abs(number));
}
```

Output:
Number : -1234
Absolute value : 1234

pow()

Header file: <MATH.H>

Power function calculates x raised to the y (x**y i.e. x to the power y)

Syntax: pow(double x, double y);

```
#include <math.h>
 #include <stdio.h>
 void main()
 {
    double x = 2.0, y = 3.0, p;
    p = pow(x, y);
    printf("%lf raised to %lf is %lf \n", x, y, p);
 }
```

Output:
2.000000 raised to 3.000000 is 8.000000

sqrt()

Header file: <MATH.H>

sqrt calculates the positive square root of the input value.

Syntax: sqrt(double x);

```
#include <math.h>
 #include <stdio.h>
 void main()
 {
    double x = 4.0, result;
    result = sqrt(x);
    printf("Square root of %lf is %lf \n", x, result);
}
```

Output:
Square root of 4.000000 is 2.000000

2. <conio.h>

clrscr()

Header file: <CONIO.H>

clrscr clears the current text window and places the cursor in the upper left-hand corner (at position 1,1).

Syntax: clrscr();

```
#include <conio.h>
 void main()
 {  clrscr();
    printf("Hello……\n”);
    clrscr();
    printf("The screen has been cleared!");
    getch();
 }
```

getch()

Header file: <CONIO.H>

getch gets a character from console but does not echo (display) to the screen.

Syntax: getch();

Return Value: Return the character read from the keyboard.

```
#include <conio.h>
#include <stdio.h>
void main()
{  char ch;
   printf("Input a character:");
   ch = getch();
}
```

ch = getch();
statement asks for input but inputted character will not be displayed on screen.

3. <ctype.h>

isalnum(), isalpha(), isdigit(), islower(), isupper(), isprint(), isspace()

Header file: <CTYPE.H>

Syntax: isalnum(c); isalpha(c); isdigit(c); islower(c); isupper(c); isprint(c); isspace(c);

Each function returns a non-zero value for true and 0 for false.

Function	To check
isalnum	c is a letter or digit (A to Z or a to z or 0 to 9)
isalpha	c is a letter (A to Z or a to z)
isdigit	c is a digit (0 to 9)
islower	c is a lowercase letter (a to z)
isupper	c is an uppercase letter (A to Z)
isprint	c is a printing character
isspace	c is a space, tab, carriage return, new line, vertical tab, or formfeed

Example:

```
#include <stdio.h>
#include <ctype.h>
void main()
{  char ch='A';
   if  (isupper(ch) != 0)
       printf("%c is uppercase character",ch);
   else
       printf("%c is not uppercase character",ch);
}
```

Output:
A is uppercase character

toupper()

Header file: <CTYPE.H>

Translate character to uppercase.

Syntax: toupper (ch);

Return Value: If ch is lowercase, toupper return its converted value. If ch is not lowercase, returns ch unchanged.

```
#include <string.h>
#include <stdio.h>
#include <ctype.h>
void main()
{
   int length, i;
   char string[20] = "This is a string";
   length = strlen(string);
   for (i=0; i<length; i++)
        string[i] = toupper(string[i]);
   printf("%s\n",string);
}
```

Output:
THIS IS A STRING

tolower()

Header file: <CTYPE.H>

Translate character to lowercase.

Syntax: tolower (ch);

Return Value: If ch is uppercase, tolower return its converted value. If ch is not uppercase, returns ch unchanged.

```
#include <string.h>
#include <stdio.h>
#include <ctype.h>
void main()
{
   int length, i;
   char string[20] = "THIS IS A STRING";
   length = strlen(string);
   for (i=0; i<length; i++)
        string[i] = tolower(string[i]);
   printf("%s\n",string);
}
```

Output:
this is a string

ARRAYS

Fundamental data type like int, float, double, char etc. are very useful in programming. But they can store only one value at a time.

In many applications we need to use large amount of data. i.e. We require to read, process and print large volume of data. C supports a derived data type known as array for such applications.

An array is a fixed-size sequenced collection of related data items that share a common name. e.g. an array with name salary can be used to represent a set of salaries of a group of employees.

***Advantages / Need of an array**

1. Single name can be used to represent a collection of items.
2. Individual elements can be referred by writing index or subscript. This way we can develop compact and efficient programs.
3. Loop can be used with the subscript as control variable to read entire array, perform calculations and print out the results.
4. Programs will become dynamic and compact.

***Disadvantages / Limitations of an array**

C performs no bounds checking. So care should be taken to ensure that the array indices are within the declared limits.

ONE-DIMENSIONAL INTEGER & FLOAT ARRAY

An **array** is a group of related data items that share a common name. For example, we can define an array name salary to represent a set of salaries of a group of employees.

A list of items can be given one variable name using only one subscript and such a variable is called a **single subscripted variable** or **one-dimensional array.** Arrays can be any variable type.

The complete set of values is referred to as an array. Individual values / data item are called **elements**.

A particular element of an array is indicated by writing a number called **index number** or **subscript** in brackets after the array name. For e.g. salary[10] represents the salary of the 11^{th} employee because in C language subscript starts with 0.

The subscript of an array can be integer constants, integer variables like i, or expressions that produce (yield) integers.

***Storage representation of One Dimensional integer and float Array**

One Dimensional integer Array

If we want to represent a set of five subject marks, say (35, 40, 20, 57, 19), by an array variable marks, then we may declare the variable number as

int marks[5];

and the computer reserves five storage locations as shown below:

marks[0]	marks[1]	marks[2]	marks[3]	marks[4]

When we assign values to array element, it will look as follows:

35	40	20	57	19
marks[0]	marks[1]	marks[2]	marks[3]	marks[4]

One Dimensional Float Array

If we want to represent height of five students, say (5.2, 5.8, 6.1, 5.7, 5.9), by an array variable height, then we may declare the variable number as

float height[5];

and the computer reserves five storage locations as shown below:

height[0]	height[1]	height[2]	height[3]	height[4]

When we assign values to array element, it will look as follows:

5.2	5.8	6.1	5.7	5.9
height[0]	height[1]	height[2]	height[3]	height[4]

***Declaration of One Dimensional integer and float Arrays**

Like any other variable, arrays must be declared before they are used. The general form of array declaration is

type variable-name[size];

The type specifies the type of element that will be contained in the array, such as int, float, or char. And the size indicates the maximum number of elements that can be stored inside the array. The size should be either a numeric constant or a symbolic constant. E.g.

int a[30]; Declares **a** to be an array containing 30 integer elements. Any subscripts from 0 to 29 are valid.

float height[50]; Declares **height** to be an array containing 50 float (real) elements. Any subscripts from 0 to 49 are valid.

***Initialization of One Dimensional integer and float Arrays**

After an array is declared, its elements must be initialized. Otherwise, they will contain "garbage".

An array can be initialized at either of the following stages:

1. **At compile time (at time of declaration of an array)**
2. **At run time (input form user and with assignment statement)**

1) Compile time initialization (at time of declaration of an array)

We can initialize the elements of an array when they are declared. The general form of initialization of arrays is:

type array-name[size] = { list of values };

The values in the list are separated by commas.

int number[3] = { 1, 5, 7 }; This will declare number as an array of size 3 and will assign values 1, 5 and 7 to element 0, 1 and 2 respectively.

float height[3] = { 5.8, 6.2, 5 } This will declare height as an array of size 3 and will assign values 5.8, 6.2 and 5.0 to element 0, 1 and 2 respectively.

If the number of values in the list is less than the number of elements, then only that many elements will be initialized. The remaining elements will be set to zero automatically.

For example, **float total[5] = { 1, 2, 3 };** will initialize the first three elements and remaining two elements to zero.

Size may be omitted. In such cases, the compiler allocates enough space for all initialized elements. For example, **float count[ ] = { 1, 1, 1 };** will declare count array to contain three elements with initial values 1 for all three elements.

Note:

1. int marks[5] = {0, 0, 0, 0, 0} **OR** int marks[5]={ 0 }; both will initialize all 5 elements of marks array with zero.
2. int marks[5] = {1, 1, 1, 1, 1} and int marks[5]={ 1 }; both are different. First will initialize all 5 elements of marks array with one but second one will initialize only 0^{th} element with value1 and rest of the elements will get value zero.

Above rules are applicable to flat array as well.

2) Run time initialization (input form user and with assignment statement)

An array can be explicitly initialized at run time.

Run time initialization can be done by

(1) Taking input from user or
(2) By assignment statement

(1) Taking input from user

We can also initialize array elements by taking input from user. This concept is known as run time initialization. This approach is usually applied for initializing large arrays. Looping can be used to input array elements. E.g.

Input 1-D integer array	**Input 1-D float array**

int marks[10];
for (i=0; i<5; i++)
{ printf (“Enter mark %d : “,i+1);
scanf (“%d”, &marks[i]);
}
In above example, for loop will be executed 5 times and asks user to input marks each time. First inputted marks will be stored in 0^{th} element of marks array, second input in 1^{st} element and so on.
	float height[10];
	for (i=0; i<5; i++)
	{ printf (“Enter height %d : “,i+1);
	scanf (“%f”, &height[i]);
	}
	In above example, for loop will be executed 5 times and asks user to input height each time. First inputted height will be stored in 0^{th} element of height array, second input in 1^{st} element and so on.

(2) By assignment statement

Elements of one dimensional array can be initialized individually also. E.g.
int number[5];
number[0] = 5;
number[1] = a * 5;
number[2]=number[0] + number[1]; etc.

This approach is time consuming and it will increase the length of program. But it can be used to change value of array elements as per the requirement.

***Use of array elements**

Array elements can be used in programs just like any other C variable. For e.g., the following are valid statements:

a = marks[0] + 10; marks[2] = x[5] + y[10];
marks[4] = marks[0] + marks[2]; value[6] = marks[i] *3;

TWO-DIMENSIONAL INTEGER & FLOAT ARRAYS

Array variables can be used to store a list of values. There can be situations where a table of values will have to be stored.

Example of two dimensional integer array

Consider following data table which shows the value of marks in three subjects by four students:

	Subject 1	Subject 2	Subject 3
Student 1	10	12	11
Student 2	15	14	16
Student 3	17	19	18
Student 4	13	12	15

The table contains total 12 values. This table can be viewed as a matrix having 4 rows and 3 columns. Each row represents marks by a particular student and each column represents marks of a particular subject. To represent a particular value in matrix, two subscripts are required.

Example of two dimensional float array

Consider following data table which shows the value of temperature in 3 cities of 3 days:

	Day 1	Day 2	Day 3
City 1	25.1	22.8	26.4
City 2	23.5	24.0	25.5
City 3	26.4	21.6	27.0

The table contains total 9 values. This table can be viewed as a matrix having 3 rows and 3 columns. Each row represents temperature of a particular city and each column represents temperature of a particular day. To represent a particular value in matrix, two subscripts are required.

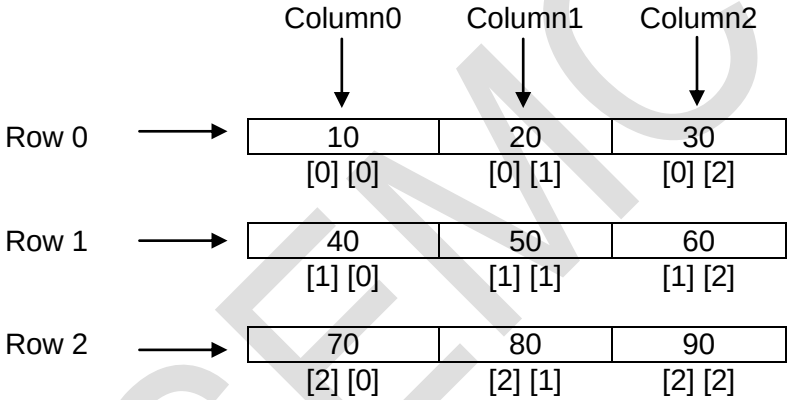
C allows user to define such table of values by using Two Dimensional Arrays. A two-dimensional array has two subscripts.

***Storage representation of Two Dimensional integer and float Array:**

Suppose we want to represent marks of 4 students in 3 subjects by an array variable test, then we may declare the variable test as

int test [3][3];

and the computer reserves 12 storage locations as shown below:



Two dimensional arrays are stored in memory as shown in the above figure. As with the single-dimensional arrays, each dimension of the array is indexed from zero to its maximum size minus one; the first index selects the row and the second index selects the column within that row.

Note: All arrays, no matter how many dimensions they have, are stored sequentially in memory.

***Declaration of Two Dimensional integer and float Array:**

Like any other variable, arrays must be declared before they are used. The general form of array declaration is

type array-name [row_size][column_size];

The type specifies the type of element that will be contained in the array, such as int, float, or char. The row_size indicates the maximum number of rows and column_size indicates the maximum number of columns that can be stored inside the array.

The size should be either a numeric constant or a symbolic constant. E.g.

int exam[4][3]; Declares **sales** to be an array containing 12 integer elements. Any subscripts from 0 to 3 are valid for row. Any subscripts from 0 to 2 are valid for columns.

float temp[3][3]; Declares **temp** to be an array containing 9 float (real) elements. Any subscripts from 0 to 2 are valid for row. Any subscripts from 0 to 2 are valid for columns.

***Initializing two-dimensional arrays**

After an array is declared, its elements must be initialized. Otherwise, they will contain "garbage".

An array can be initialized at either of the following stages:

1) At compile time (at time of declaration of an array)
2) At run time (input from user and with assignment statement)

1) Compile time initialization (at time of declaration of an array)

Like the one-dimensional arrays, two-dimensional arrays may be initialized by following their declaration with a list of initial values enclosed in braces. The general form of initialization of arrays is:

type array-name[size] = { list of values };

The values in the list are separated by commas.

int table[2][3] = {0,0,0,1,1,1}; This will declare table as an array with 2 rows and 3 columns. It will assign value 0 to 0^{th}, 1^{st} and 2^{nd} element of 0^{th} row. And it will assign value 1 to 0^{th}, 1^{st} and 2^{nd} element of 1^{st} row.

float height[2][2] = { 5.8, 6.2, 5, 5.3 } This will declare table as an array with 2 rows and 3 columns. It will assign value 5.8 and 6.2 to 0^{th} and 1^{st} element of 0^{th} row. And it will assign value 5.0 and 5.3 to 0^{th} and 1^{st} element of 1^{st} row.

Here the initialization is done row by row.

Points to remember

- The above statement can be equivalently written by surrounding the elements of each row by braces. For e.g. int table[2][3] = { {0,0,0}, {1,1,1} };
- If the values are missing in an initializer, they are automatically set to zero. For e.g.
  int table [2][3] = { {1, 1}, {2} };
  Above statement will initialize first two elements of 1^{st} row (by index 0^{th} row) to 1, first element of 2^{nd} row to 2 and all other element to 0.
- When the array is completely initialized with all values, we can omit the size of first dimension i.e. row_size. For e.g. int table [ ][3] = { {0, 0, 0}, {1, 1, 1} };

Note: Above points are applicable to float array as well.

2) Run time initialization (input from user and with assignment statement)

An array can be explicitly initialized at run time.

Run time initialization can be done by

(1) Taking input from user or
(2) By assignment statement

(1) Taking input from user

We can also initialize array elements by taking input from user. This concept is known as run time initialization. This approach is usually applied for initializing large arrays.

Looping can be used to input array elements. As two dimensional array deals with two subscripts, we need to use two for loops to control row and column subscript respectively. For E.g.

Input 2-D integer array

```
int exam[4][3];
for (i=0; i<4; i++)
    {
      for (j=0; j<3; i++)
          {
            printf ("Enter mark [%d][%d] : ",i+1,j+1);
            scanf ("%d", &exam[i][j]);
          }
     }
```

This program will input marks of 4 students in 3 subjects. For i=0, j will take value 0, 1 and 2. Thus inputting 0^{th}, 1^{st} and 2^{nd} element of 0^{th} row. i.e. Element exam[0][0], exam[0][1] and exam[0][2] will be inputted.

i will take values from 0 to 3. For each value of I, j will take value 0,1 and 2.

Input 2-D float array

```
float temp[3][3];
for (i=0; i<4; i++)
    {
      for (j=0; j<3; i++)
          {
            printf ("Enter temperature [%d][%d] : ",i+1,j+1);
            scanf ("%f", &temp[i]);
          }
     }
```

2-D float array behaves exactly as 2-D integer array.

(2) By assignment statement

Elements of two dimensional array can be initialized individually also. E.g.

```
int exam[4][3];
exam[0][2] = 15;
exam[1][1] = a * 5;
exam[2][2] = exam[2][0] + exam[2][1];  etc.
```

This approach is time consuming and it will increase the length of program. But it can be used to change value of array elements as per the requirement.

Multidimensional arrays

A two-dimensional array has a row-and-column structure. A three-dimensional array could be thought of as a cube. Four-dimensional arrays (and higher) are probably best left to your imagination.

C allows arrays of three or more dimensions. The exact limit is determined by the compiler. The general form of a multidimensional array is

type array-name [s1][s2][s3]….[sm];

Some example are: **int survey[3][5][12];**
float table[5][4][5][3];

survey is a three-dimensional array declared to contain 180 integer type elements. Similarly table is a four-dimensional array containing 300 elements of floating-point type.

The array survey may represent a survey data of rainfall during the last three years from January to December in five cities.

Disclaimer: The study material is compiled by Ami D. Trivedi. The basic objective of this material is to supplement teaching and discussion in the classroom in the subject. Students are required to go for extra reading in the subject through library work.