Session

7

Condition

Concepts

Objectives

At the end of this session, you will be able to:

- *Explain the Selection Construct*
  - The if Statement
  - The if – else statement
  - The Multi if statement
  - The Nested if statement
  - The switch Statement

Introduction

7.1 What is a conditional statement?

Conditional statements enable us to change the flow of the program. Các câu lệnh điều kiện cho phép chúng ta thay đổi luồng của chương trình. Based on a condition, a statement or a sequence of statements take alternative actions. Dựa trên một điều kiện, một câu lệnh hoặc một chuỗi các câu lệnh sẽ thực hiện các hành động thay thế.

One of the most fundamental concepts of computer science is if a certain condition is true, the computer is directed to take one course of action; and if the condition is false, it is directed to do something else. Một trong những khái niệm cơ bản nhất của khoa học máy tính là nếu một điều kiện nào đó là đúng, máy tính sẽ được hướng dẫn thực hiện một hành động; và nếu điều kiện đó sai, nó sẽ được hướng dẫn thực hiện một hành động khác.

Concepts

7.2 Selection Statements

C supports two types of selection statements:

- The `if` statement
- The `switch` statement

Let us explore these two selection statements.

7.2.1 The 'if' Statement

The if statement allows decisions to be made by checking for a given condition to be true or false. Câu lệnh if cho phép đưa ra quyết định bằng cách kiểm tra xem một điều kiện nào đó có đúng hay sai. Such conditions involve the comparison and logical operators discussed in the fourth session. Những điều kiện như vậy liên quan đến các toán tử so sánh và toán tử logic đã được thảo luận trong buổi học thứ tư.

The general form of the `if` statement is:

```
if (expression)
    statements;
```

The expression should always be enclosed in parentheses. Biểu thức nên luôn được đặt trong dấu ngoặc đơn. The statement after the keyword if is the condition (or expression) to be checked. Câu lệnh sau từ khóa if là điều kiện (hoặc biểu thức) cần được kiểm tra. This is followed by a statement or a set of statements, which are executed only if the condition (or expression) evaluates to true. Điều này được theo sau bởi một câu lệnh hoặc một tập hợp các câu lệnh, sẽ được thực hiện chỉ khi điều kiện (hoặc biểu thức) được đánh giá là đúng.

Example 1:

```
#include <stdio.h>
void main()
{
    int x, y;
    char a = 'y';
```

```
    x = y = 0;
    if (a == 'y')
    {
        x += 5;
        printf("The numbers are %d and \t%d", x, y);
    }
}
```

The above will produce an output as given below:

```
The numbers are 5 and     0
```

This is because the variable `a` has been assigned the value **y**.

Note that the block of statements following the `if` statement are enclosed in curly braces { }. Remember that if the constructs have more than one statement following it, the statements have to be treated like a block and have to be enclosed within curly braces. If the curly braces are not placed in the above case, only the first statement (x += 5) will be executed when the `if` statement evaluates to true.

Given below is an example which tests whether a given year is a leap year or not (Dưới đây là một ví dụ kiểm tra xem một năm cho trước có phải là năm nhuận hay không.). A leap year (nam nhuận) is one which is divisible by 4 or 400 but not by 100. We use an `if` statement to test this condition (Chúng ta sử dụng một câu lệnh if để kiểm tra điều kiện này.).

Example 2:

```
/* To test for a leap year */
#include <stdio.h>
void main()
{
    int year;
    printf("\nPlease enter a year :");
    scanf("%d",&year);
    if(year % 4 == 0 && year % 100 != 0 || year % 400 == 0)
        printf("\n%d is a leap year", year);
}
```

A sample output is shown below:

```
Please enter a year : 1988
1988 is a leap year
```

The condition year % 4 == 0 && year % 100 != 0 || year % 400 == 0 results in a value of 1 if the given year is a leap year. In such a case, the year followed by the message “is a leap year” is printed on the screen.

No message is printed if the condition does not hold good.

7.2.2 The 'if…else' statement

So far we have studied the simplest form of the `if` statement, which gives us a choice of executing a statement or a block of statements or skipping them. C also lets us choose between two statements by using the `if-else` structure. The formal syntax is:

```
if (expression)
    statement-1;
else
    statement-2;
```

The expression is evaluated; if it is true (non-zero), `statement-1` is executed. If it is false (zero) then `statement-2` is executed. The statements following `if` and `else` can be simple or compound. The indentation is not required but it is good programming style. It shows at a glance those statements whose execution depends on a test. (Indentation is the margin from the left side of the paper)

Example 3:

```
#include <stdio.h>
void main()
{
    int num , res ;
    printf("Enter a number :");
    scanf("%d",&num);
    res = num % 2;
    if (res == 0)
        printf("The number is Even");
    else
        printf("The number is Odd");
}
```

Example 4

```
/* To convert an uppercase character to lowercase */
#include <stdio.h>
void main()
{
    char c;
    printf("Please enter a character : ");
    scanf("%c",&c);
    if (c >= 'A' && c <= 'Z')
        printf("Lowercase character = %c", c + 'a' - 'A');
    else
        printf("Character Entered is = %c",c);
}
```

The expression `c >= 'A' && c <='Z'` tests if a letter is uppercase. If the expression evaluates to `true`, the input character is converted to lowercase using the expression `c + 'a' - 'A'`, and output using the `printf()` function. If the value of the expression is `false`, the statement following `else` is executed and the character is output without any change.

7.2.3 Multiple Choice – 'else–if ' statements

The `if` statement lets us choose whether or not to do some action. The `if-else` statement lets us choose between two actions. C also offers us more than two choices. We can extend the `if-else` structure with `else-if` to accommodate this fact. That is, the `else` part of an `if-else` statement consists of another `if-else` statement. This enables multiple conditions to be tested and hence offers several choices.

The general syntax for this is:

```
if (expression) statement;
else
if (expression) statement;
else
if (expression) statement;
.
.
else statement;
```

This construct is also known as the **if-else-if ladder** or **if-else-if staircase**.

Concepts

The above indentation is easily understandable with one or two `if`s. It can get confusing when the number of `if`s increases because in that case it gets deeply indented. So, the `if-else-if` statement is generally indented as:

```
if (expression)
    statement;
else if (expression)
    statement;
else if (expression)
    statement;
    .
    .
else
    statement;
```

The conditions are evaluated from top to down. Các điều kiện được đánh giá từ trên xuống dưới. As soon as a true condition is found, the statement associated with it is executed and the rest of the ladder is bypassed. Ngay khi một điều kiện đúng được tìm thấy, câu lệnh liên kết với nó sẽ được thực thi và phần còn lại của cấu trúc sẽ bị bỏ qua. If none of the conditions are true, then the final else is executed. Nếu không có điều kiện nào đúng, thì else cuối cùng sẽ được thực thi. If the final else is not present, no action takes place if all other conditions are false. Nếu else cuối cùng không có mặt, sẽ không có hành động nào được thực hiện nếu tất cả các điều kiện khác đều sai.

The following example takes a number from the user. Ví dụ dưới đây yêu cầu người dùng nhập một số. If the number is between 1 and 3, it prints the number; or else it prints “Invalid choice.” Nếu số nằm trong khoảng từ 1 đến 3, nó sẽ in ra số đó; nếu không, nó sẽ in ra “Lựa chọn không hợp lệ.”

Example 5:

```
#include <stdio.h>
main()
{
    int x;
    x = 0;
    clrscr ();
    printf("Enter Choice (1 - 3) : ");
    scanf("%d", &x);
    if (x == 1)
        printf ("\nChoice is 1");
    else if (x == 2)
        printf ("\nChoice is 2");
    else if (x == 3)
        printf ("\nChoice is 3");
    else
        printf ("\nInvalid Choice");
}
```

If we wish to have more than one statement following the `if` or the `else` statement, they should be grouped together between curly brackets { }. Such a grouping is called a 'compound statement' or 'a block'.

```
if (result >= 45) {
    printf("Passed\n");
    printf("Congratulations\n");
}
else {
    printf("Failed\n");
    printf("Good luck next time\n");
}
```

7.2.4 Nested if

A nested `if` is an `if` statement, which is placed within another `if` or `else` statement. In C, an `else` statement always refers to the nearest `if` statement that is within the same block as the `else` statement and is not already associated with an `if` statement. For example,

```
if (expression-1)
{
    if (expression-2)
        statement1;
    if (expression-3)
        statement2;
    else
        statement3; /* with if(expression-3) */
}
else
    statement4; /* with if(expression-1) */
```

Concepts

In the above segment, control comes to the second `if` statement if the value of `expression-1` is `true`. If `expression-2` is `true` then `statement1` is executed. The `statement2` is executed when `expression-3` is `true`, otherwise `statement3` is executed. If `expression-1` turns out to be `false` then `statement4` is executed.

Since the `else` part of the `else-if` is optional, it is possible to have constructs similar to the one given below:

```
if (condition-1)
    if (condition-2)
       statement-1;
    else
        statement-2;
next statement;
```

In the above segment of code, if `condition-1` turns out to be `true`, then the control is transferred to the second `if` statement and `condition-2` is evaluated. If it is `true`, then `statement-1` is executed, otherwise `statement-2` is executed, followed by the execution of the `next statement`. If `condition-1` is `false`, then control passes directly to `next statement`.

Consider an example where marks1 and marks2 are the scores obtained by a student in two subjects. Ví dụ này xem xét hai điểm số marks1 và marks2, là điểm số mà một sinh viên đạt được trong hai môn học. Grace marks of 5 are added to marks2 if marks1 is greater than 50 but marks2 is less than 50. Nếu marks1 lớn hơn 50 nhưng marks2 nhỏ hơn 50, 5 điểm thưởng sẽ được cộng vào marks2. If marks2 is greater than or equal to 50, then the grade obtained by the student is ‘A’. Nếu marks2 lớn hơn hoặc bằng 50, thì điểm số của sinh viên là 'A'. This condition can be coded using the if construct as follows: Điều kiện này có thể được mã hóa bằng cách sử dụng cấu trúc if như sau:

```
if (marks1 > 50 && marks2 <50)
    marks2 = marks2 + 5;
if (marks2 >=50)
    grade = ‘A’;
```

Some users may be tempted to code as follows:

```
if (marks1 > 50 )
if (marks2 <50)
    marks2 = marks2 + 5;
else
    grade = ‘A’;
```

In this segment, **‘A’** will be assigned to grade only when marks1 is greater than 50 and marks2 is greater than or equal to 50. But, according to our requirement, the grade should be assigned the value **‘A’** after testing and adding grace marks to marks2. Also the grade is independent of the value in marks1.

Because the `else` part of an `if` is optional, it is not clear when an `else` is omitted from a nested `if`

sequence. This is resolved by using the rule that the `else` is associated with the closest previous if not associated with an `else`.

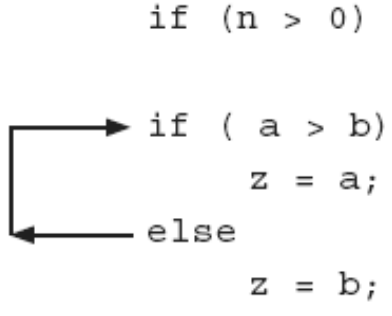
For example, in

```
if (n > 0)
    if (a > b)
        z = a;
    else
        z = b;
```

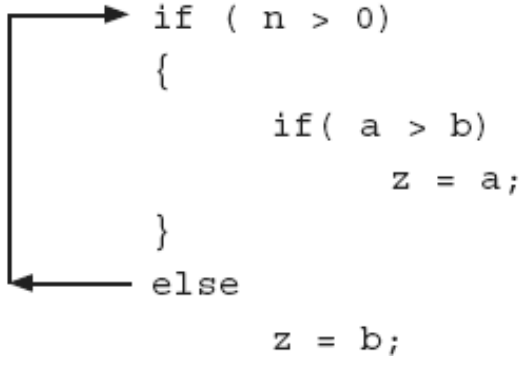
the `else` goes with the inner `if`. The indentation is a way of showing this relationship. However, indentation does not associate an `else` with an `if`. If that is not what you want curly braces { } must be used to force the proper association.

```
if (n > 0)
{
    if (a > b)
        z = a;
}
else
    z = b;
```

The following figure shows the association between `if` and `else` in a sequence of nested `if` statements.



`else` goes with the most recent `if`



`else` goes with the first `if` since braces enclose the inner `if` statement

Concepts

As an example of nested `if`, consider the following:

Example 6:

```
#include <stdio.h>
void main ()
{
    int x, y;
    x = y = 0;
    clrscr ();
    printf ("Enter Choice (1 - 3) :" );
    scanf ("%d", &x);
    if (x == 1)
    {
        printf("\nEnter value for y (1 - 5) :" );
        scanf ("%d", &y);
        if (y <= 5)
            printf("\nThe value for y is : %d", y);
        else
            printf("\nThe value of y exceeds 5" );
    }
    else
        printf ("\nChoice entered was not 1");
}
```

A complete program below shows the usefulness of nested `if`.

A company manufactures three products namely computer stationery, fixed disks and computers. The following codes are used to indicate them.

Product	Code
Computer Stationery	1
Fixed Disks	2
Computers	3

Condition

The company has a discount policy as follows:

Product	Order Amount	Discount Rate
Computer Stationery	$ 500/- or more	12 %
Computer Stationery	$ 300/- or more	8 %
Computer Stationery	below $300 /-	2 %
Fixed Disks	$ 2000/- or more	10 %
Fixed Disks	$ 1500/- or more	5 %
Computers	$ 5000/- or more	10 %
Computers	$ 2500/- or more	5 %

The program to calculate the discount amounts as per this policy is given below:

Example 7:

```
#include <stdio.h>
void main()
{
    int productcode;
    float orderamount, rate = 0.0 ;
    printf("\n Please enter the product code :" );
    scanf("%d", &productcode);
    printf("Please enter the order amount :" );
    scanf(" %f" , &orderamount);
    if (productcode == 1)
    {
        if (orderamount >= 500 )
            rate = 0.12;
        else if (orderamount >= 300 )
            rate = 0.08;
        else
            rate = 0.02;
    }
    else if ( productcode == 2)
    {
        if (orderamount >= 2000)
            rate = 0.10 ;
        else if (orderamount >= 1500)
            rate = 0.05;
    }
```

```
    else if ( productcode == 3)
    {
        if (orderamount >= 5000)
            rate = 0.10 ;
        else if (orderamount >= 2500)
            rate = 0.05;
    }
    orderamount - = orderamount * rate;
    printf( "The net order amount is % .2f \n", orderamount);
}
```

A sample output is shown below:

```
Please enter the product code : 3
Please enter the order amount : 6000
The net order amount is 5400
```

Observe that the last `else` in a sequence of `else-if`'s need not check any condition specifically. For example, if the product code is 1 and the order amount is less than $ 300, there was no need to check, since all the possibilities were already taken care of. The output of the above program for a product code of 3 and order amount of $ 6000 is as shown above:

Modify the above program to take care of a condition where the input consists of an invalid product code. This can be easily achieved by adding an `else` to the sequence of if's which are testing the product code. If such a product code is encountered, the program should terminate without calculating the net order amount.

7.2.5 The 'switch' Statement

The `switch` statement is a multi-way decision maker that tests the value of an expression against a list of integer or character constants. When a match is found, the statements associated with that constant are executed. The general syntax of the `switch` statement is:

```
switch (expression)
{
    case constant1:
        statement sequence
        break;
    case constant2:
        statement sequence
        break;
```

```
    case constant3:
        statement sequence
        break;
    default:
        statement sequence;
}
```

where, `switch`, `case` and `default` are the keywords and a statement sequence can be simple statement or a compound statement, which need not be enclosed in parentheses. The expression following `switch` must be enclosed in parentheses, and the body of the `switch` must be enclosed within curly braces { }. The data type of the expression to be evaluated and the data type of the case constants specified should be matching. As suggested by the name, case labels can be only an integer or a character constant. It can also be constant expressions provided the expressions do not contain any variable names. All the case labels must be different.

In the `switch` statement, the expression is evaluated, and the value is compared with the case labels in the given order. If a label matches with the value of the expression, the statement mentioned against that case label will be executed. The `break` statement (discussed later) ensures immediate exit from the `switch` statement. If a `break` is not used, the statements in the following case labels are also executed irrespective of whether that case value is satisfied or not. This execution will continue till a `break` is encountered. Therefore, `break` is said to be one of the most important statement while using the `switch` statement.

The statements written against default will be executed, if none of the other cases are satisfied. The default statement is optional. If it is not present, and the value of the expression does not match with any of the cases, then no action will be taken The case labels and default may be in any order .

Consider the following example.

Example 8:

```
#include <stdio.h>
main ()
{
    char ch;
    clrscr();
    printf ("\nEnter a lower cased alphabet (a - z) :" );
    scanf("%c", &ch);
    if(ch < 'a' || ch > 'z')
        printf("\nCharacter not a lower cased alphabet");
    else
        switch(ch)
```

```
        {
            case 'a' :
            case 'e' :
            case 'i' :
            case 'o' :
            case 'u' :
                printf("\nCharacter is a vowel");
                break;
            case 'z' :
                printf ("\nLast Alphabet (z) was entered");
                break;
            default :
                printf("\nCharacter is a consonant");
                break;
        }
}
```

The program will take a lower cased alphabet as an input and display whether it is a vowel, the last alphabet or any of the other alphabets. If any other key, other than lower cased alphabets, is entered, the message "Character not a lower cased alphabet" is displayed.

It is recommended that a `break` be put even after the last case or default even though it is logically unnecessary. This is helpful when another case gets added at the end.

Here's another example where the expression in `switch` is an integer variable and each of the case labels is an integer.

Example 9:

```
/* Integer constants as case labels */
#include <stdio.h>
void main()
{
    int basic;
    printf("\n Please enter your basic:" );
    scanf("%d",&basic);
    switch (basic)
    {
        case 200 : printf("\n Bonus is dollar %d\n", 50);
        break;
```

```
        case 300 : printf("\n Bonus is dollar %d\n", 125);
            break;
        case 400 : printf("\n Bonus is dollar %d\n", 140);
            break;
        case 500 : printf("\n Bonus is dollar %d\n", 175);
            break;
        default : printf("\n Invalid entry");
            break;
    }
}
```

Concepts

Summary

- Conditional statements enable us to change the flow of the program.
- C supports two types of selection statements (câu lệnh): if and switch ().
- The following are some of the conditional statements.

  - The if statement – A condition is checked; (Câu lệnh if – Một điều kiện được kiểm tra; n if the results turn out to be true, the statements following it are executed and joins the main program. (Nếu kết quả đúng, các câu lệnh theo sau sẽ được thực thi và nối vào chương trình chính.) If the results are false, it joins the main program straight. (Nếu kết quả sai, nó sẽ nối vào chương trình chính ngay lập tức.)
  - The if…else statement – A condition is checked; (Câu lệnh if…else – Một điều kiện được kiểm tra; if the results are true, the statements following the if statement are executed. (Nếu kết quả đúng, các câu lệnh theo sau câu lệnh if sẽ được thực thi.) If the results are false, then the statements following the else part are executed. (Nếu kết quả sai, thì các câu lệnh theo sau phần else sẽ được thực thi.)
  - Nested if statements – Nested if statements comprise of an if within an if statement. (Câu lệnh if lồng nhau – Các câu lệnh if lồng nhau bao gồm một câu lệnh if nằm bên trong một câu lệnh if khác.)
  - The switch statement is a special multi-way decision maker that tests whether an expression matches one of a number of constant values. (Câu lệnh switch là một bộ quyết định đa chiều đặc biệt kiểm tra xem một biểu thức có khớp với một trong số các giá trị hằng hay không.) It branches accordingly. (Nó phân nhánh tương ứng.)

Check Your Progress

1. _____ statements enable us to change the flow of a program.

   A. Conditional B. Loop

   C. Sequence D. None of the above

2. The `else` statement is optional. **(T/F)**

3. A _____ is an `if` statement, which is placed within another `if` or `else`.

   A. Multi `if` B. Nested `if`

   C. `switch`ed `if` D. None of the above

4. The _____ statement is a multi-way decision maker that tests the value of an expression against a list of integer or character constants.

   A. Sequence B. `if`

   C. `switch` D. None of the above

5.
```
if (expression)
        statement 1
else
        statement 2
```

   Which statement will be executed when expression is `false`?

   A. `statement 1` B. `statement 2`

Concepts

Try It Yourself

1. Write a program that accepts two numbers **a** and **b** and checks whether or not **a** is divisible by **b**.

2. Write a program to accept 2 numbers and tell whether the product of the two numbers is equal to or greater than 1000.

3. Write a program to accept 2 numbers. Calculate the difference between the two values.

   If the difference is equal to any of the values entered, then display the following message:

   Difference is equal to value <number of value entered>

   If the difference is not equal to any of the values entered, display the following message:

   Difference is not equal to any of the values entered

4. Montek company gives allowances to its employees depending on their grade as follows:

Grade	Allowance
A	300
B	250
Others	100

   Calculate the salary at the end of the month. (Accept Salary and Grade from the user)

5. Write a program to evaluate the Grade of a student for the following constraints:

   If marks > 75 – grade A

   If 60 < marks < 75 – grade B

   If 45 < marks<60 – grade C

   If 35 < marks<45 - grade D

   If marks < 35 – grade E

Session

8 Condition (Lab)

Lab Guide

Objectives

At the end of this session, you will be able to:

- *Use :*
  - *The if statement*
  - *The if – else statement*
  - *The Multi if statement*
  - *The Nested if statement*
  - *The switch statement*

8.1 The 'if' statement

In this section you will write a program to calculate the commission given to salesmen depending on their sales amount. (Trong phần này, bạn sẽ viết một chương trình để tính toán hoa hồng được trao cho các nhân viên bán hàng dựa trên số tiền doanh thu của họ.)

Problem

SARA Company gives a `10%` commission to its salesmen if their monthly sales amount to `$10,000` or more. Calculate the commission at the end of the month.

The program declares two '`float`' variables **`sales_amt`** and **`com`**. Note that the variables are declared in the same line of code using a comma (`,`) to separate one from the other.

Consider the following line of code :

```
printf("Enter the Sales Amount : ");
scanf("%f", &sales_amt);
```

In the `printf()` function, we display a message to enter the sales amount, and in the `scanf()` function

we use %f to accept a value from the user. The value you are entering goes to the variable **sales_amt**.

```
if (sales_amt >= 10000)
    com = sales_amt * 0.1;
```

The above statement is used to check whether the value in **sales_amt** variable is greater than or equal to **10000** . >= is a comparison operator which gives you a **true** or **false** value. In this case, if you are entering a value **15000** the condition (sales_amt >= 10000) is **true**. If **true**, it will execute the statement com = sales_amt * 0.1 . Now the value of **com** will be **1500**. If the condition is **false**, it will print the commission value as **0**. Here we can see that if condition has only one statement. If there is more than one statement for an if condition, give the statements in curly braces { }.

```
printf("\n Commission = %f",com);
```

The above statement is used to display the value of commission. '%f' is used to display the value of the 'float' variables mentioned after the comma at the end of printf(). Thus, printf() displays the commission amount of simple interest.

8.1.1 Calculating the commission

1. **Create a new file.**

2. **Type the following code in the 'Edit Window' :**

```
#include <stdio.h>
#include <conio.h>
void main()
{
    float com=0,sales_amt;
    clrscr();
    printf("Enter the Sales Amount : ");
    scanf("%f",&sales_amt);
    if (sales_amt >= 10000)
        com = sales_amt * 0.1;
    printf("\n Commission = %f",com);
}
```

3. **Save the file with the name comm.c**

4. **Compile the file comm.c**

5. **Execute the program `comm.c`**

6. **Return to the editor.**

OUTPUT :

```
Enter the Sales Amount : 15000
Commision = 1500.0000
```

8.2 The 'if- else' statement

In this section you will write a program that makes use of the `if-else` statement. The program displays the greater of the two given numbers.

```
if(num1 > num2)
    printf("\n The greater number is : %d", num1);
else
    printf("\n The greater number is : %d", num2);
```

In this program the first `printf()` is executed only if the value of the variable **num1** is greater than the value of the variable **num2**, in which case the `else` part is ignored. If the value of the variable **num1** is not greater than the value of the variable **num2**, the first `printf()` is ignored. In this case the second `printf()`, the one following the `else`, is executed.

In this program, since the value of **num1** is greater than **num2**, the first `printf()` is executed.

```
#include <stdio.h>
#include <conio.h>
void main()
{
    int num1,num2,;
    clrscr();
    num1 = 540;
    num2 = 243;
```

```
    if (num1 > num2)
        printf( "\n The greater number is : %d", num1);
    else
        printf("\n The greater number is : %d", num2);
}
```

3.	**Save the file with the name `ifelse.c`**
4.	**Compile the file `ifelse.c`**
5.	**Execute the program `ifelse.c`**
6.	**Return to the editor.**

OUTPUT :

```
The greater number is : 540
```

8.3 The 'if – else – if' statement

you will write a program that makes use of the `if – else – if` statement. The program displays the greater of the two given numbers or displays that the numbers are equal.

In the previous program, there are two 'integer' variables **`num1`** and **`num2`** declared. The variables are assigned a value.

```
if(num1 == num2)
    printf("\n Numbers are Equal");
else if(num1 < num2)
    printf("\n The Larger Number is : %d", num2);
else
    printf("\n The Larger Number is : %d", num1);
```

In this program, the first '`if`' condition `(num1==num2)` checks if the value of the variable **`num2`** is equal to the value of the variable **`num1`**. In C, the `==` symbol is used to check if the two operands are equal. If the first condition `(num1==num2)` is `true` then, the following `printf()` will be executed and the control will come out of the loop. If the first condition is not true, the `else if` condition will be checked. In case this condition `(num1<num2)` is satisfied, the associated `printf()` will be executed and the control will come out of the loop. If neither the `if` nor the `else if` conditions are satisfied, then the `else` condition

will be executed.

In this program, since the value of **num1** is lesser than **num2**, the second `printf()` statement is executed.

```
#include <stdio.h>
#include <conio.h>
void main()
{
    int num1 , num2 ;
    num1 = 77;
    num2 = 90;
    if( num1 == num2)
        printf("\n The Numbers are equal");
    else if (num1 < num2)
        printf("\n The Larger Number is : %d", num2);
    else
        printf("\n The Larger Number is : %d", num1);
}
```

3. **Save the file with the name `ifelseif.c`**

4. **Compile the file `ifelseif.c`**

5. **Execute the program `ifelseif.c`**

6. **Return to the editor.**

OUTPUT :

```
The Larger Number is: 90
```

8.4 The 'Nested if' statement

MONTEK company has decided to give commission to their sales department depending on the sales amount. The commission structure is defined as follows:

Sales	Grade	Commission
> 10,000 $	A	10 %
	—	8%
<=10,000 $	—	5 %

Calculate the commission at the end of each month.

In this program we are calculating the commission based on grade and sales amount.

Consider the following lines of code:

```
printf("Enter the Sales Amount : ");
scanf("%f",&sales_amt);
printf("\n Enter the Grade : ");
scanf("%c",&grade);
```

The first `scanf()` is used to accept the sales amount, and second `scanf()` is used to accept the grade. `%c` format specifier is used to accept a single character from the user.

```
if (sales_amt > 10000)
    if (grade == 'A')
        com = sales_amt * 0.1;
    else
        com = sales_amt * 0.08;
else
    com = sales_amt * 0.05;
```

Suppose we enter the sales amount as **15000** and grade as '**A**' . It will check the `if` condition (`sales_amt > 10000);` since this condition is **true** it will go to the second `if (grade == 'A')`. This condition is also **true**, so it will calculate commission `com = sales_amt * 0.1`.

We will see another situation with sales amount **15000** and grade as '**B**'. It will check the first `if` condition `(sales_amt > 10000)`, in this case it is **true**. Then it will go to the second `if` statement, in this case

it is not true, it will go to the respective `else` condition i.e.,

```
else
    com = sales_amt * 0.08;
```

If we enter a value **10000** or less, it will go the last `else` condition and calculate the commission as

`com = sales_amt * 0.05;`

```
#include <stdio.h>
#include <conio.h>
void main()
{
    float com=0,sales_amt;
    char grade;
    clrscr();
    printf("Enter the Sales Amount : ");
    scanf("%f", &sales_amt);
    printf("\n Enter the Grade : ");
    scanf("%c", &grade);
    if (sales_amt > 10000)
        if (grade == 'A')
            com = sales_amt * 0.1;
        else
            com = sales_amt * 0.08;
    else
        com = sales_amt * 0.05;
    printf("\n Commission = %f", com);
}
```

3. **Save the file with the name `nestif.c`**

4. **Compile the file `nestif.c`**

5. **Execute the program `nestif.c`**

6. **Return to the editor.**

Lab Guide

OUTPUT :

```
Enter the Sales Amount : 15000
Enter the grade : A
Commission = 1500
```

8.5 Using the 'switch' statement

In this section you will use the 'switch' statement. The program displays appropriate result depending on the mathematical operator used.

In this program there are two integer variables **num1** and **num2** and a character variable **op** declared. Values are assigned to the variables. A mathematical operator is stored in the variable **op**.

The variable **op** is passed in the expression following the 'switch'. The first case compares the value of the variable **op** to be '+'. If the label (+) matches with the value in **op**, then the following lines of code are executed:

```
res = num1 + num2;
printf("\n The sum is : %d", res);
break;
```

The sum of **num1** and **num2** is stored in the variable **res**. The printf() displays the value of the variable **res**. The 'break' statement makes an exit from the 'switch' statement.

In the second case the value of **op** is '–' , then num1 – num2 is executed. Similarly if the value of **op** is '*' and '/' , num1 *num2 and num1 / num2 is executed.

If none of the above cases are satisfied, then the printf() of the 'default' is executed.

```
#include <stdio.h>
#include <conio.h>
void main()
{
    int num1, num2 ,res;
    char op;
    num1 = 90;
    num2 = 33;
    op = '-';
```

```
    clrscr();
    switch(op)
    {
        case '+':
            res = num1 + num2;
            printf("\n The Sum is : %d", res);
            break;
        case '-':
            res = num1 - num2;
            printf("\n Number after Subtraction : %d", res);
            break;
        case '/':
            res = num1 / num2;
            printf("\n Number after Division : %d", res);
            break;
        case '*':
            res = num1 * num2;
            printf("\n Number after multiplication : %d", res);
            break;
        default:
            printf("\n Invalid");
        break;
    }
}
```

3. Save the file with the name `case.c`

4. Compile the file `case.c`

5. Execute the program `case.c`

6. Return to the editor.

OUTPUT :

```
Number after subtraction: 57
```

Part II : For the next 30 Minutes:

1. A student appears for a test in 3 subjects (Một học sinh tham gia một bài kiểm tra ở 3 môn). Each test is out of 100 marks (Mỗi bài kiểm tra được tính trên 100 điểm). The percentage of each student has to be calculated and depending on the percentage calculated, grades are given as under(Phần trăm của mỗi học sinh phải được tính toán và tùy thuộc vào phần trăm được tính toán, điểm số được đưa ra như sau) :

Percentage	Grade
>=90	E +
80 - < 90	E
70 - < 80	A +
60 - < 70	A
50 - < 60	B +
< 50	FAIL

To do this :

a. Accept the marks of 3 subjects and store in 3 different variables say **M1**, **M2** and **M3**.

b. Calculate the percentage (`per = (M1 + M2 + M3) / 3`)

c. Check the grade depending on the percentage calculated.

d. Display the grade.

Try It Yourself

1. Declare two variables **x** and **y**. Assign values to these variables. Number **x** should be printed only if it is less than **2000** or greater than **3000**, and number **y** should be printed only if it is between **100** and **500**.

2. Write a program to show your computer's capabilities. The user types in a letter of the alphabet and your program should display the corresponding language or package available. Some sample input and output is given below :

Input	Output
A or a	Ada
B or b	Basic
C or c	COBOL
D or d	dBASE III
f or F	Fortran
p or P	Pascal
v or V	Visual C++

Using the '`switch`' statement (sử dụng câu lệnh switch) to choose and display (để chọn và hiển thị) the appropriate message (thông báo phù hợp). Use the `default` label to display a message (Sử dụng nhãn mặc định để hiển thị thông báo) if the input does not match any of the above letters (nếu đầu vào không khớp với bất kỳ chữ cái nào ở trên.).

3. Accept values in three variables and print the highest value (Chấp nhận giá trị trong ba biến và in ra giá trị cao nhất.).


“ It is always in season
for old men to learn ”

Session

9 Loop

Concepts

Objectives

At the end of this session, you will be able to:

- *Understand the 'for' loop in C*
- *Work with the 'comma' operator*
- *Understand nested loops*
- *Understand the 'while' loop and the 'do-while' loop*
- *Work with 'break' and 'continue' statements*
- *Understand the 'exit()' function*

Introduction

9.1 The Loop Structure

A loop is a section of code in a program which is executed repeatedly, until a specific condition is satisfied. (Một vòng lặp là một đoạn mã trong một chương trình được thực thi lặp đi lặp lại, cho đến khi một điều kiện cụ thể được thỏa mãn.) The loop concept is fundamental to structured programming. (Khái niệm vòng lặp là cơ bản trong lập trình có cấu trúc.)

The loop structures available in C are:

- The `for` loop
- The `while` loop
- The `do…while` loop

The condition which controls the execution of the loop is coded using the Relational and Logical operators in C. (Điều kiện điều khiển việc thực thi vòng lặp được mã hóa bằng cách sử dụng các toán tử so sánh và logic trong C.)

Concepts

9.1.1 The 'for' Loop

The general syntax of the `for` loop is:

```
for(initialize counter; conditional test; re-evaluation parameter)
{
    Statement(s);
}
```

The **initialize counter** is an assignment statement that sets the loop control variable, before entering the loop. This statement is executed only once. The **conditional test** is a relational expression, which determines when the loop will exit. The **re-evaluation parameter** defines how the loop control variable changes (mostly, an increment or decrement in the variable set at the start) each time the loop is repeated. These three sections of the `for` loop must be separated by semicolons. The statement, which forms the body of the loop, can either be a single statement or a compound statement (more than one statement).

The `for` loop continues to execute as long as the conditional test turns out to be **`true`**. When the condition becomes **`false`**, the program resumes with the statement following the `for` loop.

Consider the following program:

Example 1:

```
/*This program demonstrates the for loop in a C program */
#include <stdio.h>
main()
{
    int count;
    printf("\tThis is a \n");
    for(count = 1; count <=6 ; count++)
        printf("\n\t\t nice");
    printf("\n\t\t world. \n");
}
```

The sample output is shown below :

```
This is a
    nice
    nice
    nice
    nice
```

```
    nice
    nice
    world.
```

Look at the `for` loop section in the program.

1. The initialization parameter is `count = 1`

   It is executed only once when the loop is entered, and the variable **`count`** is set to **`1`**.

2. The conditional test is `count <=6`

   A test is made to determine whether the current value of **`count`** is less than or equal to **`6`**. If the test is **`true`**, then the body of the loop is executed.

3. The body of the loop consists of a single statement (Thân của vòng lặp bao gồm một câu lệnh duy nhất)

   ```
   printf("\n\n\t nice");
   ```

   This statement can be enclosed in braces to make the body of the loop more visible. (Câu lệnh này có thể được đặt trong dấu ngoặc nhọn để làm cho phần thân của vòng lặp de nhìn thấy hơn.)

4. The re-evaluation parameter is **`count++`** , increments the value of **`count`** by **`1`** for the next iteration.

The steps 2,3,4 are repeated until the conditional test becomes **`false`**. The loop will be executed 6 times for value of **`count`** ranging from **`1`** to **`6`**. Hence, the word **`nice`** appears six times on the screen. For the next iteration, **`count`** is incremented to **`7`**. Since this value is greater than **`6`**, the loop is ended and the statement that follows it is executed.

The following program prints even numbers from 1 to 25 (Chương trình sau đây in ra các số chẵn từ 1 đến 25).

Example 2:

```
#include <stdio.h>
main()
{
    int num;
    printf("The even numbers from 1 to 25 are:\n\n");
    for(num=2; num <= 25; num+=2)
        printf ("%d\n",num);
}
```

Concepts

The output for the above code will be:

```
The even numbers from 1 to 25 are:
2
4
6
8
10
12
14
16
18
18
20
22
24
```

The `for` loop above initializes the integer variable **`num`** to **`2`** (to get an even number) and increments it by **`2`** every time the loop is executed (Vòng lặp for ở trên khởi tạo biến số nguyên num thành 2 (để có được một số chẵn) và tăng nó lên 2 mỗi lần vòng lặp được thực thi.).

In `for` loops, the conditional test is always performed at the top of the loop (Trong vòng lặp for, kiểm tra điều kiện luôn được thực hiện ở đầu vòng lặp). This means that the code inside the loop is not executed if the condition is **`false`** in the beginning itself (Điều này có nghĩa là mã bên trong vòng lặp không được thực thi nếu điều kiện là sai ngay từ đầu).

The 'Comma' operator

The scope of the `for` loop can be extended by including more than one initializations or increment expressions in the `for` loop specification. The format is :

```
exprn1 , exprn2
```

The expressions are separated by the 'comma' operator and evaluated from left to right. (Các biểu thức được ngăn cách bởi toán tử 'phẩy' và được đánh giá từ trái sang phải.) The order of the evaluation is important if the value of the second expression depends on the newly calculated value of exprn1. (Thứ tự đánh giá là quan trọng nếu giá trị của biểu thức thứ hai phụ thuộc vào giá trị vừa được tính toán của exprn1.) This operator has the lowest precedence among the C operators. (Toán tử này có độ ưu tiên thấp nhất trong số các toán tử C.)

The following example, which prints an addition table for a constant result, will illustrate the concept of the comma operator clearly: (Ví dụ sau đây, in bảng cộng cho một kết quả cố định, sẽ minh họa rõ ràng khái niệm về toán tử phẩy:)

Example 3:

```
/* program illustrates the use of the comma operator */
#include <stdio.h>
```

```
main()
{
    int i, j , max;
    printf("Please enter the maximum value \n");
    printf("for which a table can be printed: ");
    scanf("%d", &max);
    for(i = 0 , j = max ; i <=max ; i++, j--)
        printf("\n%d + %d = %d",i, j, i + j);
}
```

A sample output is shown below :

```
Please enter the maximum value -
for which a table can be printed: 5
0 + 5 = 5
1 + 4 = 5
2 + 3 = 5
3 + 2 = 5
4 + 1 = 5
5 + 0 = 5
```

Note that in the `for` loop, the initialization parameter is

```
i = 0, j = max
```

When it is executed , **`i`** is assigned the value **`0`** and **`j`** is assigned the value present in **`max`**.

The re-evaluation (increment) parameter again consists of two expressions:

```
i++, j--
```

after each iteration, **`i`** is incremented by **`1`** and **`j`** is decremented by **`1`**. The sum of these two variables which is always equal to **`max`** is printed out.

- **'Nested for' Loops**

  A `for` loop is said to be nested when it occurs within another `for` loop. The code will be something like this:

Concepts

```
for(i = 1; i < max1; i++)
{
     .
     .
     for(j = 0; j < = max2; j++)
     {
          .
          .
     }
     .
     .
}
```

Consider the following example,

Example 4:

```
#include <stdio.h>
main()
{
     int i, j, k;
     i = 0;
     printf("Enter no. of rows :");
     scanf("%d", &i);
     printf("\n");
     for(j = 0; j < i ; j++)
     {
          printf("\n");
          for (k = 0; k <= j; k++) /*inner for loop */
             printf("*");
     }
}
```

This program displays '*' on each line and for each line increments the number of '*' to be printed by 1. It takes the number of rows for which '*' has to be displayed as input. For example, if the input is 5, the output will be

```
*
**
***
****
*****
```

➢ **More on 'for' loops**

The `for` loop can be used without one or all of its definitions.

For example,

```
.
.
for(num = 0; num ! = 255;)
{
    printf("Enter no. ");
    scanf("%d", &num);
.
.
}
```

The above will accept a value for **num** until the input is **255**. This loop does not have any re-evaluation factor. The loop terminates when **num** becomes **255**.

Similarly, consider

```
.
.
printf("Enter value for checking :");
scanf("%d", &num);
for(; num < 100; )
{
    .
    .
}
```

This loop does not have an initialization factor or a re-evaluation factor.

The `for` loop, when used without any definitions, gives an infinite loop.

Concepts

```
for( ; ; )
    printf("This loop will go on and on and on ...\n");
```

However, a `break` statement used within this loop will cause an exit from it.

```
.
.
for( ; ; )
{
    printf("This will go on and on");
    i = getchar();
    if(i == 'X' || i == 'x')
        break;
}
.
.
```

The above loop will run until the user enters **x** or **X**.

The `for` loop (or any other loop- hoặc bất kỳ vòng lặp khác) can also be used without the body (statements). This helps to increase the efficiency of some algorithms and to create time delay loops (Điều này giúp tăng hiệu quả của một số thuật toán và tạo ra vòng lặp trễ thời gian).

```
for(i = 0; i < xyz_value, i++);
```

is an example of a time delay loop.

9.1.2 The 'while' Loop

The second kind of loop structure in C is the `while` loop. Its general syntax is:

```
while(condition is true)
    statement;
```

Where, statement is either an empty statement, a single statement, or a block of statements. (Ở đây, câu lệnh có thể là một câu lệnh rỗng, một câu lệnh đơn, hoặc một khối các câu lệnh.) If a set of statements are present in a while loop then they must be enclosed inside curly braces { }. (Nếu một tập hợp các câu lệnh có mặt trong một vòng lặp while, thì chúng phải được đặt trong dấu ngoặc nhọn { }.) The condition may be any expression. (Điều kiện có thể là bất kỳ biểu thức nào.) The loop iterates while this condition is true. (Vòng lặp lặp lại trong khi điều kiện này đúng.) The program control is passed to the line after the loop code when the condition becomes false. (Quyền điều khiển chương trình được chuyển đến dòng sau mã vòng lặp khi điều kiện trở thành sai.)

The for loop can be used provided the number of iterations are known before the loop starts executing. (Vòng lặp for có thể được sử dụng nếu số lần lặp được biết trước khi vòng lặp bắt đầu thực thi.) When the number of iterations to be performed is not known beforehand, the while loop can be used. (Khi số lần lặp cần thực hiện không được biết trước, vòng lặp while có thể được sử dụng.)

Example 5:

```
/* A simple program using the while loop */
#include <stdio.h>
main()
{
    int count = 1;
    while( count <= 10)
    {
        printf("\n This is iteration %d\n",count);
        count++;
    }
    printf("\n The loop is completed. \n");
}
```

A sample output is shown below:

```
This is iteration 1
This is iteration 2
This is iteration 3
This is iteration 4
This is iteration 5
This is iteration 6
This is iteration 7
This is iteration 8
This is iteration 9
This is iteration 10
The loop is completed.
```

The program initially (chương trình ban đầu) sets (đặt) the value of **count (giá trị của count)** to **1(là 1)** in the declaration statement itself (trong chinhs câu lệnh khai báo). The next statement executed is the `while` statement. The condition is first checked. The current value of **count** is **1**, which is less than **10**. The condition test results is **true**, and therefore the statements in the body of the `while` loop are executed. Therefore, they are enclosed within curly braces `{ }`. The value of **count** becomes **2** after 1st iteration. Then the condition is checked again. This process is repeated until the value of **count** becomes greater than **10**. When the loop is exited, the second `printf()` statement is executed.

Like `for` loops, `while` loops check (kiểm tra) the condition (điều kiện) at the top of the loop (ở đầu vòng lặp). This means that (có nghĩa rằng) the loop code (mã vòng lặp) is not executed (không được thực thi), if the condition (nếu điều kiện) is **false** at the start (sai từ đầu).

The conditional test in the loop may be as complex as required. (Kiểm tra điều kiện trong vòng lặp có thể phức tạp theo yêu cầu.) The variables in the conditional test may be reassigned values within the loop, but a point to remember is that eventually the condition test must become false otherwise the loop will never end. (Các biến trong kiểm tra điều kiện có thể được gán lại giá trị trong vòng lặp, nhưng một điểm cần nhớ là cuối cùng kiểm tra điều kiện phải trở thành sai, nếu không vòng lặp sẽ không bao giờ kết thúc.

Concepts

The following is an example of an infinite while loop. (Dưới đây là một ví dụ về vòng lặp while vô hạn.)

Example 6:

```
#include <stdio.h>
main ()
{
    int count = 0;
    while(count < 100)
    {
        printf("This goes on forever, HELP!!!\n");
        count+=10;
        printf("\t%d", count);
        count-=10;
        printf("\t%d", count);
        printf("\nCtrl-C will help");
    }
}
```

In the above, count is always 0, which is less than 100 and so the expression always returns a true value. (Trong ví dụ trên, count luôn là 0, nhỏ hơn 100 và vì vậy biểu thức luôn trả về giá trị đúng.) Hence the loop never ends. (Do đó, vòng lặp không bao giờ kết thúc.)

If more than one condition is to be checked to end a while loop, the loop will end if at least one of the conditions becomes false. (Nếu có hơn một điều kiện cần được kiểm tra để kết thúc vòng lặp while, vòng lặp sẽ kết thúc nếu ít nhất một trong các điều kiện trở thành sai.) The following example illustrates this. (Ví dụ sau đây minh họa điều này.)

Example 7:

```
#include <stdio.h>
main()
{
    int i, j;
    i = 0;
    a = 10;
    while(i < 100 && a > 5)
    {
        .
        .
        i++;
        a-= 2;
    }
```

```
    .
    .
}
```

This loop will perform **3** iterations, the first time **a** will be **10**, the next time it will be **8** and the third time it will be **6**. After this though **i** is still less than **100** (**i** is **3**), **a** takes the value **4**, and the condition `a>5` turns out to be false, so the loop ends.

Let us write a useful program. It accepts input from the console and prints it on the screen.

The program ends when you press **^Z (Ctrl+Z)**.

Example 8:

```
/* ECHO PROGRAM */
/* A program to accept input data from the console and print it on the
screen */
/* End of input data is indicated by pressing '^Z' */
# include <stdio.h>
main()
{
    char ch;
    while((ch = getchar()) != EOF)
    {
        putchar(ch);
    }
}
```

A sample output is shown below :

```
Have
Have
a
a
good
good
day
day
^ z
```

Concepts

The user input (đầu vào của ng dùng) is highlighted (được tô sáng). After entering (sau khi nhập) a set of characters (tập hợp các kí tự), the contents will get echoed back to the screen only when you press **<Return>**. This is because (điều này bởi vì) the characters (kí tự) you enter through the keyboard (bạn nhập qua bàn phím) gets stored (được lưu trữ) in the keyboard buffered input (bộ đệm đầu vào của bàn phím) and the `putchar()` statement (câu lệnh putchar) fetches it (sẽ lấy nó) from the buffer (từ bộ nhớ đệm) when you press **<Return>**. Notice how the input is terminated (được kết thúc) with a **^Z**, which is the end of file character on MS DOS operating system (là kí tự kết thúc trên hệ điều hành MS DOS).

9.1.3 The 'do .... while' Loop

The `do...while` loop is sometimes referred to as the `do` loop in C. Unlike `for` and `while` loops, this loop checks its condition at the end of the loop, that is after the loop has been executed. This means that the `do.... while` loop will execute at least once, even if the condition is false at first.

The general syntax of the `do... while` loop is:

```
do {
    statement;
} while (condition);
```

The curly brackets (dấu ngoặc nhọn) are not necessary (k cần thiết) when only one statement (khi chỉ có 1 câu lệnh) is present within the loop (trong vòng lặp), but it is a good habit (nhưng nó là thói quen tốt) to use them (khi sd chúng). The `do...while` loop (vòng lặp do...while) iterates until (lặp lại cho đến khi) the condition (điều kiện) becomes false. In the `do...while` loop the statement (vòng lặp câu lệnh do...while) (block of statements-khối câu lệnh) is executed first (thực hiện trước), then the condition is checked.If it is **true**, control (điều khiển) is transferred (được chuyển đến) to the `do` statement (câu lệnh do). When the condition becomes **false**, control is transferred (chuyển) to the statement after the loop.

Example 9:

```
/*accept only int values */
#include <stdio.h>
main ()
{
    int num1, num2;
    num2 = 0;
    do{
        printf( "\nEnter a number : ");
        scanf("%d",&num1);
        printf( " No. is %d",num1);
        num2++;
    }while(num1 != 0);
    printf("\nThe total numbers entered were %d",--num2);
```

```
    /* num2 is decremented before printing because count for last integer (0)
is not to be considered */
}
```

A sample output is shown below:

```
Enter a number : 10
No. is 10
Enter a number : 300
No. is 300
Enter a number : 45
No. is 45
Enter a number : 0
No. is 0
The total numbers entered were 3
```

The above code will accept integers and display them until **zero** (0) is entered. It will then, exit from the **do...while** loop and print the number of integers entered.

- **'Nested while' and 'do...while' Loops**

  Like `for` loops, `while` and `do...while` loops can also be nested. An example is given below:

  Example 10:

```
#include <stdio.h>
main()
{
    int x;
    char i, ans;
    i = ' ';
    do
    {
        clrscr ();
        x = 0;
        ans = 'y';
        printf("\nEnter sequence of character:");
        do {
            i = getchar ();
            x++;
```

```
        }while (i != '\n');
        i = ' ';
        printf("\nNumber of characters entered is: %d", --x);
        printf("\nMore sequences (Y/N) ?");
        ans = getch ();
    }while(ans == 'Y' || ans == 'y');
}
```

A sample output is shown below:

```
Enter sequence of character:Good Morning !
Number of characters entered is: 14
More sequences (Y/N) ? N
```

This program code first asks the user to enter a sequence of characters till the enter key is hit (nested while). Once the **Enter** key is pressed, the program exits from the inner `do…while` loop. The program then asks the user if more sequences of characters are to be entered. If the user types '**y**' or '**Y**', the outer `while` condition is true and the program prompts the user to enter another sequence. This goes on till the user hits any other key except '**y**' or '**Y**'. The program then ends.

9.2 Jump Statements

C has four statements that perform an unconditional branch: `return`, `goto`, `break`, and `continue`. Unconditional branching means the transfer of control from the point where it is, to a specified statement. Of the above jump statements, `return` and `goto` can be used anywhere in the program, whereas `break` and `continue` statements are used in conjunction with any of the loop statements.

9.2.1 The 'return' Statement

The `return` statement is used to return from a function. It causes execution to return to the point at which the call to the function was made. The `return` statement can have a value with it, which it returns to the program. The general syntax of the `return` statement is:

```
return expression ;
```

The expression is optional. More than one return can be used in a function. However the function will return when the first return is met. The `return` statement will be clear after a discussion on functions.

9.2.2 The 'goto' Statement

Though C is a structured programming language, it contains the following unstructured forms of program control:

- `goto`
- `label`

A `goto` statement transfers control not only to any other statement within the same function in a C program, but it allows jumps in and out of blocks. Therefore, it violates the rules of a strictly structured programming language.

The general syntax of the `goto` statement is,

```
goto label;
```

where **`label`** is an identifier which must appear as a prefix to another C statement in the same function. The semicolon(`;`) after the label identifier marks the end of the `goto` statement. `goto` statements in a program make it difficult to read. They reduce program reliability and make the program difficult to maintain. However, they are used because they can provide useful means of getting out of deeply nested loops. Consider the following code:

```
for(...){
    for(...) {
        for(...) {
            while(..) {
                if(...) goto error1;
                    .
                    .
                    .
                }
            }
        }
    }
error1: printf("Error !!!");
```

As seen, the label appears as a prefix to another statement in the program.

```
label: statement
```

or

Concepts

```
label: {
    statement sequence
}
```

Example 11:

```
# include <stdio.h>
#include <conio.h>
main()
{
    int num ;
    clrscr();
    label1:
        printf("\nEnter a number (1) :");
        scanf("%d",&num);
        if(num==1)
            goto Test;
        else
            goto label1:
    Test:
        printf("All done…");
}
```

A sample output is given below:

```
Enter a number : 4
Enter a number : 5
Enter a number : 1
All done...
```

9.2.3 The 'break' Statement

The `break` statement has two uses. It can be used to terminate a case in the `switch` statement and/or to force immediate ending of a loop, bypassing the normal loop conditional test.

When the `break` statement is met inside a loop, the loop is immediately ended and the program control is passed to the statement following the loop. For example,

Example 12:

```
#include <stdio.h>
main ()
{
    int count1, count2;
    for(count1 = 1, count2 = 0; count1 <=100; count1++)
    {
        printf("Enter %d Count2 : ", count1);
        scanf("%d", &count2);
        if(count2 == 100) break;
    }
}
```

A sample output is shown below:

```
Enter 1 count2 : 10
Enter 2 count2 : 20
Enter 3 count2 : 100
```

In the above code, the user can enter **100** values for **j**. However, if **100** is entered, the loop ends and control is passed to the next statement.

Another point to be remembered while using a `break` is that it causes an exit from an inner loop. This means that if a `for` loop is nested within another `for` loop, and a `break` statement is encountered in the inner loop, the control is passed back to the outer `for` loop.

9.2.4 The 'continue' Statement

The `continue` statement causes the next iteration of the enclosing loop to being. When this statement is met in the program, the remaining statements in the body of the loop are skipped and the control is passed to the re-initialization step.

In case of the `for` loop, `continue` causes the increment portions of the loop and then the conditional test to be executed. In case of the `while` and `do...while` loops, program control passes to the conditional tests. For example:

Example 13:

```
#include <stdio.h>
main ()
```

```
{
    int num;
    for(num = 1; num <=100; num++)
    {
        if (num % 9 == 0) continue;
        printf("%d\t", num);
    }
}
```

The above prints all numbers from **1** to **100**, which are not divisible by **9**. The output will look somewhat like this.

```
1 2 3 4 5 6 7 8 10 11 12 13 14 15 16 17
19 20 21 22 23 24 25 26 28 29 30 31 32 33 34 35
37 38 39 40 41 42 43 44 46 47 48 49 50 51 52 53
55 56 57 58 59 60 61 62 64 65 66 67 68 69 70 71
73 74 75 76 77 78 79 80 82 83 84 85 86 87 88 89
91 92 93 94 95 96 97 98 100
```

9.3 The 'exit()' function

The `exit()` function is a standard C library function. Its working is similar to the working of a jump statement, the major difference being that the jump statements are used to break out of a loop, whereas `exit()` is used to break out of the program. This function causes immediate ending of the program and control is transferred back to the operating system. The `exit()` function is usually used to check if a mandatory condition for a program execution is satisfied or not. The general syntax of the `exit()` function is

```
exit(int return_code);
```

where, **return_code** is optional. **Zero** is generally used as a **return_code** to indicate normal program ending. Other values indicate some sort of error.

Summary

- The loop structures available in C are:
  - The for loop
  - The while loop
  - The do...while loop
- The for loop enables repeated execution statements in C. It uses three expressions, separated by semicolons, to control the looping process. The statement part of the loop can be simple statement or a compound statement.
- The 'comma' operator is occasionally useful in the for statements . Of all the operators in C it has the lowest priority.
- The body of a do statement is executed at least once.
- C has four statements that perform an unconditional branch : return, goto, break, and continue.
- The break statement enables early exit from a simple or a nesting of loops. The continue statement causes the next iteration of the loop to begin.
- A goto statement transfers control to any other statement within same function in a C program, but it allows jumps in and out of blocks.
- The exit() function causes immediate termination of the program and control is transferred back to the operating system.

Check Your Progress

1. _____ allows a set of instructions to be performed until a certain condition is reached.

   A. Loop B. Structure

   C. Operator D. None of the above

2. _____ loops check the condition at the top of the loop which means the loop code is not executed, if the condition is false at the start.

   A. `while` loop B. `for` loop

   C. `do..while` loop D. None of the above

3. A _____ is used to separate the three parts of the expression in a `for` loop.

   A. comma B. semicolon

   C. hyphen D. None of the above

4. The _____ loop checks its condition at the end of the loop, that is after the loop has been executed.

   A. `while` loop B. `for` loop

   C. `do..while` loop D. None of the above

5. The _____ statement causes execution to return to the point at which the call to the function was made.

   A. `exit` B. `return`

   C. `goto` D. None of the above

6. The _____ statement violates the rules of a strictly structured programming language.

   A. `exit` B. `return`

Check Your Progress

C. `goto` D. None of the above

7. The _____ function causes immediate termination of the program and control is transferred back to the operating system

A. `exit` B. `return`

C. `goto` D. None of the above

Concepts

Try It Yourself

1. Write a program to print the series 100, 95 , 90, 85,……., 5.

2. Accept two numbers num1 and num2. Find the sum of all odd numbers between the two numbers entered.

3. Write a program to generate the Fibonacci series. (1,1,2,3,5,8,13,…….).

4. Write a program to display the following patterns.

```
(a) 1
    12
    123
    1234
    12345
```

```
(b) 12345
    1234
    123
    12
    1
```

5. Write a program to generate the following pattern.

```
*******
******
*****
****
***
**
*
```

Session

10 Loop (Lab)

Lab Guide

Objectives

At the end of this session, you will be able to:

- *Use the loop structure*
- *Write Some Programs*
  - Using 'for' Loop
  - Using 'while' loop
  - Using 'do…while' loop

The steps given in this session are detailed, comprehensive and carefully thought through. This has been done so that the learning objectives are met and the understanding of the tool is complete. Follow the steps carefully.

Part I – For the first 1 Hour and 30 Minutes :

10.1 Using the 'for' Loop

In this section you will write a program using the '`for`' loop. The program displays even numbers from 1 to 30.

In the program an 'integer' variable, **`num`**, is declared. The '`for`' loop is used to display the even numbers till 30. The first argument of the '`for`' loop, initializes the variable **`num`** to 2. The second argument of the '`for`' loop, checks whether the value of the variable is less than or equal to 30. If this condition is satisfied the statement in the loop is executed. The '`printf()`' statement is used to display the value of the variable **`num`**.

In the third argument, the value of the variable **`num`** is incremented by 2. In C, **`num += 2`** is like **`num = num + 2`**. The '`printf()`' is executed until the second argument is satisfied. Once the value of the variable becomes greater than 30 the condition does not get satisfied and hence the loop is not executed. The curly brackets are not necessary when only one statement is present within the loop, but it is a good programming practice to use them.

Session 10

Loop (Lab)

1. Create a new file.
2. Type the following code in the 'Edit window' :

```
#include <stdio.h>
#include <conio.h>
void main()
{
    int num;
    clrscr();
    printf("\n The even Numbers from 1 to 30 are \n ");
    for (num = 2 ; num <= 30 ; num += 2)
       printf("%d\n",num);
}
```

3. Save the file with the name, `for.c`
4. Compile the file, `for.c`
5. Execute the program, `for.c`.
6. Return to the editor.

OUTPUT :

```
The even Numbers from 1 to 30 are
2
4
6
8
10
12
14
16
18
20
22
24
26
28
30
```

10.2 Using the 'while' Loop

In this section you will write a program using the '`while`' loop. The program displays numbers from 10 to 0 in the reverse order.

In this program there is an integer variable **`num`**. The variable is initialized to a value.

Consider the following lines of code:

```
while (num >= 0)
{
    printf("\n%d",num);
    num--;
}
```

The 'while' statement checks, if the value of the variable **`num`** is greater than 0. If the condition is satisfied the 'printf()' is executed and the value of the variable **`num`** is reduced by 1. In C, **`num--`** woks as **`num = num - 1`**. The 'while' loop continues till, the value of the variable is greater than 1 or equal to 0.

1. Create a new file.

2. Type the following code in the 'Edit Window':

```
#include <stdio.h>
#include <conio.h>
void main()
{
    int num;
    clrscr();
    num = 10;
    printf("\n Countdown");
    while(num >= 0)
    {
      printf("\n%d",num);
      num--;
    }
}
```

Lab Guide

3. Save the file with the name, **while.c**.

4. Compile the file, **while.c** .

5. Execute the program, **while.c**

6. Return to the editor.

OUTPUT:

```
Countdown
10
9
8
7
6
5
4
3
2
1
0
```

10.3 Using do-while Loop

In this section you will write a program that makes use of the 'do-while' loop. The `do` loop differs from the `while` loop only in that it executes the statement before evaluating the expression. An important consequence of this, which you should keep in mind, is that unlike a `while` loop, the contents of a `do` loop will be executed at least once. Because the `while` loop evaluates the expression before executing the statement, if the condition is `false` (zero) right at the beginning, the statement will never be executed.

The program will accept integers and display them until zero (0) is entered. It will then exit from the `do…while` loop and print the number of integers entered.

The program declares two variables **cnt** and **cnt1**. Inside the `do-while` loop we are entering the number by giving the code below:

```
printf("\nEnter a Number : ");
scanf("%d", &cnt);
```

The code below will display the number entered.

```
printf("No. is %d",cnt);
```

cnt1++ will increment the value of **cnt1** by 1. Suppose if we are entering the number as 0 ,first it will print the value and then check the condition . In this case the condition is `true`. It will come out of the loop and print the value of **cnt1**. **cnt1** is decremented before printing because count for last integer (0) is not to be considered.

1. Create a new file.

2. Type the following code in the 'Edit Window':

```
#include <stdio.h>
#include <conio.h>
void main()
{
    int cnt, cnt1 ;
    clrscr();
    cnt = cnt1 = 0;
    do
    {
      printf("\nEnter a Number : ");
      scanf("%d", &cnt);
      printf("No. is %d", cnt);
      cnt1++;
    } while ( cnt != 0 );
    printf("\n The total numbers entered were %d", --cnt1);
}
```

3. Save the file with the name , dowhile.c.

4. Compile the file, dowhile.c.

5. Execute the program, dowhile.c.

6. Return to the editor.

OUTPUT:

```
Enter a number 11
No is 11

Enter a number 50
No is 50

Enter a number 0
No is 0

The total numbers entered were 2
```

10.4 Using the break statement

The `break` causes immediate exit from a `for`, `while`, `do/while`, or `switch` statement.

The following program demonstrates the use of `break` statement.

Consider the following code:

```
for(cnt = 1; cnt <= 10 ; cnt++)
{
    if (cnt == 5)
      break;
    printf("%d\n",cnt);
}
```

The above code, uses a `for` loop to print the values from 1 to 10. The value of **`cnt`** is initialized to 1 in the beginning, then it will check the condition. If the condition is `true` it will execute the statements inside the `for` loop.

In this case the program prints only 1, 2, 3 and 4 when the value of **`cnt`** becomes 5, the if condition becomes `true` and control will come out of the loop.

1. Create a new file.

2. Type the following code in the 'Edit Window':

```
#include <stdio.h>
#include <conio.h>
void main()
{
    int cnt ;
    clrscr();
    for(cnt =1 ; cnt <=10 ; cnt++)
    {
      if(cnt==5)
        break;
      printf("%d\t", cnt);
    }
}
```

3. **Save the file with the name, `breakex.c`.**

4. **Compile the file, `breakex.c`.**

5. **Execute the program, `breakex.c`.**

6. **Return to the editor.**

OUTPUT:

```
1    2    3    4
```

10.5 Using the continue statement

The `continue` statement when executed in a `while`, `for`, or `do/while` causes all other statements in that control statement to be skipped and the next iteration to be performed

The following program demonstrates the use of `continue` statement.

Consider the following code:

```
for(cnt = 1; cnt <= 10; cnt++)
{
    if(cnt == 5)
      continue;
    printf("%d\n", cnt);
}
```

The above code uses a `for` loop to print the values from 1 to 10. The value of **`cnt`** is initialized to 1 in the beginning , then it will check the condition. If the condition is `true` it will execute the statements inside the `for` loop.

In this case, the program prints only 1, 2, 3, 4, 6, 7, 8, 9 and 10. When the value of **`cnt`** becomes 5, the if condition becomes `true` and it goes back to the `for` loop again without printing the value 5.

1. Create a new file.

2. Type the following code in the 'Edit Window':

```
#include <stdio.h>
#include <conio.h>
void main()
{
    int cnt ;
    clrscr();
    for(cnt = 1; cnt <= 10; cnt++)
    {
      if(cnt==5)
        continue;
      printf("%d\t", cnt);
    }
}
```

3. Save the file with the name, `breakex.c`.

4. Compile the file, `breakex.c`.

5. Execute the program, `breakex.c`.

6. Return to the editor.

OUTPUT:

```
1    2    3    4    6    7    8    9    10
```

Part II – For the next 30 Minutes:

1. Find the factorial of a number.

   For example,

   - $n! = n * (n-1) * (n-2) * ..... * 1$
   - $4! = 4 * 3 * 2 * 1$
   - $1! = 1$
   - $0! = 1$

   Hint:

   1. Get the number.
   2. To begin, set the factorial of the number to be one.
   3. While the number is greater than one.
   4. Set the factorial to be the factorial multiplied by the number.
   5. Decrement the number.
   6. Print out the factorial.

Try It Yourself

1. Declare a variable which has the age of the person. Print the user's name as many times as his age.

2. Write a program to generate the following pattern:

   1

   1 2

   1 2 3

   1 2 3 4

   1 2 3 4 5

   1 2 3 4 5 6

   1 2 3 4 5 6 7

   1 2 3 4 5 6 7 8

   1 2 3 4 5 6 7 8 9

3. Write a program to print a multiplication table for a given number.

Session

11

Arrays - MẢNG

Concepts

Objectives

At the end of this session, you will be able to:

- *Explain (giải thích) array elements (phần tử mảng) and indices (chỉ số)*
- *Define (định nghĩa) an array*
- *Explain (giải thích) array handling (cách xử lý mảng) in C*
- *Explain how an array is initialized (được khởi tạo)*
- *Explain string / character arrays*
- *Explain two dimensional arrays (Giải thích mảng hai chiều)*
- *Explain initialization of multidimensional arrays (Giải thích khởi tạo mảng đa chiều)*

Introduction

It may be difficult to store (tập hợp) a collection of similar (tương tự) data elements (phần tử dữ liệu) in different variable (trong các biến khác) .

An array (mảng) is a collection of data elements (tập hợp của các phần tử dữ liệu) of the same type (cùng kiểu). Each element of the array (mỗi phần tử của dữ liệu) has the same data type (có cùng kiểu dữ liệu), same storage class (cùng lớp bộ nhớ) and same characteristics (cùng đặc trưng). Each element (mỗi phần tử) is stored (được lưu trữ) in successive locations (vị trí liên tiếp) of the main store (của bộ nhớ chính). These elements (các phần tử này) are known as (được gọi là) **members** of the array (thành viên của mảng).

11.1 Array Elements and Indices các phần tử và chi so mảng

Each member of an array (mỗi thành viên của mảng) is identified by (được xác định bằng) an unique **index (1 chỉ số)** or **subscript (chỉ số dưới)** assigned to it (được gán cho nó). The **dimension** of an array is determined by the number of indices needed to uniquely identify each element. An index (chỉ số) is a positive integer (số nguyên dương) enclosed in square brackets [ ] (trong dấu ngoặc vuông) placed immediately after the array name (được đặt ngay sau tên mảng), without a space in between (không chứa khoảng trắng ở giữa). An **index** holds integer values starting with zero. Thus, an array **`player`** with 11 elements will be represented as,

```
player[0], player[1], player[2],.... player[10].
```

As is seen, the array element starts with **`player[0]`**, and so the last element is **`player[10]`** and not **`player[11]`**. This is because in C, an array index starts from 0; and so for an array of **`N`** elements, the last element has an index of **`N-1`**. The limits of the allowed index values are called the **bounds** of the

Concepts

array index, the **Lower** bound and the **Upper** bound. A valid index must have an integer value either between the bounds or equal to one of them. The term **valid** is used for a very specific reason. In C, if the user tries to access an element outside the legal index range (like `player[11]` in the above example of an array), it will not generate an error as far as the C compiler is concerned. However, it may access some value which can lead to unpredictable results. There is also a danger of overwriting data or program code. Hence the programmer has to ensure that all indices are within the valid bounds.

Defining an Array

An array has some particular characteristics and has to be defined using them. These characteristics include:

- The **storage** class
- The **data type** of the elements of the array
- The **array name** which indicates the location of the first member of the array
- Array size, a constant which is an integer expression evaluating to a positive value

An array is defined in the same way as a variable is defined, except that the array name is followed by one or more expressions, enclosed within square brackets, [ ], specifying the array dimension. The general syntax of an array definition is:

```
storage_class data_type array_name[size_expr]
```

Here, **`size_expr`** is an expression denoting the number of members in the array and must evaluate to a **positive integer**. Storage class is optional. The array defaults to class **automatic** for arrays defined within a function or a block and **external** for arrays defined outside a function. The array **`player`** will therefore, be declared as

```
int player[11];
```

Remember that while defining the array, the array size will be 11, though the indices of individual members of the array will range from 0 to 10.

The rules for forming array names are the same as those for variable names. An **array name** and a **variable name** cannot be the same as it leads to ambiguity. If such a declaration is given in a program, the compiler displays an error message.

- **Some Norms with Arrays**

  - All elements of an array are of the same type. This means that if an array is declared of type `int`, it cannot contain elements of any other types.

- Each element of an array can be used wherever a variable is allowed or required.
- An element of an array can be referred using a variable or an integer expression. The following are valid references:

```
player[i]; /* Where i is a variable, though care has to be
taken that i is within the range of the
defined subscript for the array player.*/
player[3] = player[2] + 5;
player[0] += 2;
player[i/2+1];
```

- Arrays can have their data type as `int`, `char`, `float`, or `double`.

11.2 Array handling in C

An array is treated differently from a variable in C. Two arrays, even if they are of the same type and size, cannot be tested for equality. Moreover, it is not possible to assign one array directly to another. Instead each array element has to be assigned separately to the corresponding element of another array. Values cannot be assigned to an array as a whole, except at the time of initialization. However, individual elements can not only be assigned values but can also be compared.

```
int player1[11], player2[11];
for( i=0; i<11; i++)
    player1[i] = player2[i];
```

The same can also be attained by using separate assignment statements as follows:

```
player1[0] = player2[0];
player1[1] = player2[1];
.
.
player1[10] = player2[10];
```

The `for` construct is the ideal way of manipulating arrays.

Example 1:

```
/* Program demonstrates a single dimensional array */
#include <stdio.h>
void main()
```

Concepts

```
{
    int num[5];
    int i;
    num[0] = 10;
    num[1] = 70;
    num[2] = 60;
    num[3] = 40;
    num[4] = 50;
    for(i=0;i<5;i++)
        printf("\n Number at [%d] is %d" ,i, num[i]);
}
```

The output is shown below:

```
Number at [0] is 10
Number at [1] is 70
Number at [2] is 60
Number at [3] is 40
Number at [4] is 50
```

The example given below accepts values into an array of size 10 and displays the highest and average values.

Example 2:

```
/* Input values are accepted from the user into the array ary[10]*/
#include <stdio.h>
void main()
{
    int ary[10];
    int i, total, high;
    for(i=0; i<10; i++)
    {
        printf("\n Enter value: %d : ", i+1);
        scanf("%d",&ary[i]);
    }
    /* Displays highest of the entered values */
    high = ary[0];
    for(i=1; i<10; i++)
```

```
    {
        if(ary[i] > high)
        high = ary[i];
    }
    printf("\nHighest value entered was %d", high);
    /* prints average of values entered for ary[10] */
    for(i=0,total=0; i<10; i++)
        total = total + ary[i];
    printf("\nThe average of the elements of ary is %d",total/i);
}
```

A sample output is shown below:

```
Enter value: 1 : 10
Enter value: 2 : 20
Enter value: 3 : 30
Enter value: 4 : 40
Enter value: 5 : 50
Enter value: 6 : 60
Enter value: 7 : 70
Enter value: 8 : 80
Enter value: 9 : 90
Enter value: 10 : 10
Highest value entered was 90
The average of the elements of ary is 46
```

- **Array Initialization**

  Automatic arrays cannot be initialized, unless each element is given a value separately. Automatic arrays should not be used without proper initialization, as the results in such cases are unpredictable. This is because the allocated storage locations assigned to the array are not automatically initialized. Whenever elements of such an array are used in arithmetic expressions, the previously existing values will be used, which are not guaranteed to be the same type as the array definition, unless the array elements are very clearly initialized. This is true not only for arrays but also for ordinary variables.

  In the following code snippet, the array elements have been assigned values using a `for` loop.

```
int ary[20], i;
for(i=0; i<20; i++)
    ary[i] = 0;
```

Concepts

Initializing an array using the `for` loop can be done using a constant value or values which are in arithmetic progression.

The `for` loop can also be used to initialize an array of alphabets as follows:

Example 3:

```
#include <stdio.h>
void main()
{
    char alpha[26];
    int i, j;
    for(i=65,j=0; i<91; i++,j++)
    {
        alpha[j] = i;
        printf("The character now assigned is %c \n", alpha[j]);
    }
    getchar();
}
```

The partial output of the above code is:

```
The character now assigned is A
The character now assigned is B
The character now assigned is C
.
.
```

The above program assigns ASCII character codes to the elements of the character array **`alpha`**. This, when printed using the `%c` specifier, results in a series of characters printed one after the other. Arrays can be initialized when they are defined. This can be done by assigning a list of values separated by commas and enclosed in curly braces { } to the array name. The values within the curly braces are assigned to the elements of the array in the order of their appearance.

For example,

```
int deci[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
static float rates[4] = {0.0, -2.5, 13.75, 18.0};
char company[5] = {'A', 'P', 'P', 'L', 'E'};
int marks[100] = {15, 13, 11, 9}
```

Arrays **deci**, **company** and **marks** are assumed to be external arrays by virtue of their position in the program. The initializing values must be constants and cannot be variables or expressions. The first few elements of the array will be initialized if the number of initializing values is fewer than the defined array dimension. The remaining elements will be initialized to zero. For example, in array **marks** after the above initialization, the first four elements (0 through 3) are initialized to 15, 13, 11 and 9 respectively. The rest of the elements are initialized to the value zero. It is not possible to initialize only elements 1 through 4, or 2 through 4, or 2 through 5 when initialization is done at the time of declaration. There is no facility in C to specify repetition of an initializing value.

In case of an explicit initialization, `class extern` or `static`, the array elements are guaranteed (unlike class `auto`) to be initialized to zero.

It is not necessary to declare the size of the array that is being initialized. If the size of the array is omitted, the compiler will compute the array size by counting the initializing values. For example, the following **external** array declaration will assess the dimension of the array **ary** to be 5 as there are five initializing values.

```
int ary[] = {1, 2, 3, 4, 5};
```

- **Strings / Character Arrays**

A **string** can be defined as a character type array, which is terminated by a NULL character. Each character of a string occupies one byte and the last character of a string is always the character '\0'. This `'\0'` is called the **null character**. It is an escape sequence like `'\n'` and stands for a character with a value of 0 (zero). As '\0' is always the last character of a string, character arrays have to be one character longer than the longest string they are required to hold. For example, an array, for example **ary**, that holds a 10- character string should be declared as:

```
char ary[11];
```

The additional location is used for storing the null character. Remember that the ending character (null character) is very important.

The string values can be input using the `scanf()` function. For the string **ary** declared above, the code statement to accept the value will be as follows:

```
scanf("%s",ary);
```

In the above statement, **ary** defines the location where the successive characters of the string will be stored.

Concepts

Example 4:

```
#include <stdio.h>
void main()
{
    char ary[5];
    int i;
    printf("\n Enter string :" );
    scanf("%s",ary);
    printf("\n The string is %s \n\n", ary);
    for (i=0; i<5; i++)
        printf("\t%d", ary[i]);
}
```

The outputs for various inputs are as follows:

If the entered string is **appl**, the output will be:

```
The string is appl
97 112 112 1 08 0
```

The input for the above is of 4-characters (**appl**) and the 5th character is the null character. This is clear from the ASCII codes for the characters printed in the second line. The fifth character is printed as 0, which is the value of the null character.

If the entered string is **apple**, the output will be:

```
The string is apple
97 112 112 108 101
```

The above output is for an input of 5 characters namely a, p, p, l, & e. This is not considered a string because the 5th character of this array is not \0. Again, this is clear from the line giving the ASCII codes of the characters a, p, p, l, e.

If the entered string is **ap**, the output will be as shown below.

```
The string is ap
97 112 0 6 100
```

In the above example, when only two characters are entered the third character is the null character. This actually indicates that the string has ended. The remaining characters are unpredictable

characters.

In the above case, the importance of the null character becomes clear. The null character actually specifies the end of the string and is the only way in which functions that work with the string will know where the end of string is.

Although C does not have a string data type, it allows string constants. A string constant is a list of characters enclosed in double quotes(“ ”). Like any other constant, it cannot be altered in the program. An example is,

```
"Hi Aptechite!"
```

The `null` character is added automatically at the end of the string by the C compiler.

C supports a wide range of string functions, which are found in the standard header file `string.h`. A few of these functions are given in Table 11.1. The working of these functions will be discussed in Session 17.

Name	Function
`strcpy(s1, s2)`	Copies **`s2`** into **`s1`**
`strcat(s1, s2)`	Joins **`s2`** onto the end of **`s1`**
`strlen(s1)`	Returns the length of **`s1`**
`strcmp(s1, s2)`	Returns 0 if **`s1`** and **`s2`** are the same; less than 0 if **`s1<s2`**; greater than 0 if **`s1>s2`**
`strchr(s1, ch)`	Returns a pointer to the first occurrence of **`ch`** in **`s1`**
`strstr(s1, s2)`	Returns a pointer to the first occurrence of **`s2`** in **`s1`**

Table 11.1: A few String functions in C

11.3 Two Dimensional Arrays

So far, we have dealt with arrays that were **single-dimensional** arrays. This means that the arrays had only one subscript. Arrays can have more than one dimension. **Multidimensional** arrays make it easier to represent multidimensional objects, such as a graph with rows and columns or the screen of the monitor. Multidimensional arrays are defined in the same way as single-dimensional arrays, except that a separate pair of square brackets [ ], is required in case of a two dimensional array. A three dimensional array will require three pairs of square brackets and so on. In general terms, a multidimensional array can be represented as

```
storage_class data_type ary[exp1][exp2]....[expN];
```

where **`ary`** is an array having a storage class **`storage_class`**, data type **`data_type`**, and **`exp1`**, **`exp2`**, up to **`expN`** are positive integer expressions that indicate the number of array elements associated with

Concepts

each subscript.

The simplest and most commonly used form of multidimensional arrays is the two-dimensional array. A two dimensional array can be thought of as an array of two 'single-dimensional' arrays. A typical two-dimensional array is the airplane or railroad timetable. To locate the information, the required row and column is determined, and information is read from the location at which they (the row and column) meet. In the same way, a two-dimensional array is a grid containing rows and columns in which each element is uniquely specified by means of its row and column coordinates. A two-dimensional array **tmp** of type `int` with 2 rows and 3 columns can be defined as,

```
int tmp[2][3];
```

This array will contain 2 X 3 (6) elements and they can be represented as:

		Column		
		0	**1**	**2**
Row	**1**	e1	e2	e3
	2	e4	e5	e6

where **e1 – e6** represent the elements of the array. Both rows and columns are numbered from 0 onwards. The element **e6** is designated by row 1 and column 2. To access this element the representation will be:

```
tmp[1][2];
```

- **Initialization of Multidimensional Arrays**

  A multidimensional array definition can include the assignment of initial values. Care must be taken regarding the order in which the initial values are assigned to the array elements (only external and static arrays can be initialized). The elements of the first row of two-dimensional array will be assigned values first, and then the elements of the second row, and so on. Consider the following array definition:

  ```
  int ary[3][4] ={1,2,3,4,5,6,7,8,9,10,11,12};
  ```

  The result of the above assignment will be as follows:

  ```
  ary[0][0] = 1    ary[0][1] = 2    ary[0][2] = 3    ary[0][3] = 4
  ary[1][0] = 5    ary[1][1] = 6    ary[1][2] = 7    ary[1][3] = 8
  ary[2][0] = 9    ary[2][1] = 10   ary[2][2] = 11   ary[2][3] = 12
  ```

  Note that the first subscript ranges from 0 to 2 and the second subscript ranges from 0 to 3. A point to remember is that array elements will be stored in neighbouring (side by side) locations in memory. The above array **ary** can be thought of as an array of 3 elements, each of which is an

array of 4 integers, and will appear as,

Row 0				Row 1				Row 2			
1	2	3	4	5	6	7	8	9	10	11	12

The natural order in which the initial values are assigned can be altered by forming groups of initial values enclosed within braces. Consider the following initialization.

```
int ary [3][4] ={
{1, 2, 3},
{4, 5, 6},
{7, 8, 9}
};
```

The array will be initialized as follows

```
ary[0][0]=1    ary[0][1]=2    ary[0][2]=3    ary[0][3]=0
ary[1][0]=4    ary[1][1]=5    ary[1][2]=6    ary[1][3]=0
ary[2][0]=7    ary[2][1]=8    ary[2][2]=9    ary[2][3]=0
```

An element of a multidimensional array can be used as a variable in C provided the proper number of subscripts is given with the array element.

Example 5:

```
/* Program to accept numbers in a two dimensional array. */
#include <stdio.h>
void main()
{
    int arr[2][3];
    int row,col;
    for(row=0;row<2;row++)
    {
        for(col=0;col<3;col++)
        {
             printf("\nEnter a Number at [%d][%d] : ",row,col);
             scanf("%d",&arr[row][col]);
         }
    }
```

Concepts

```
    for(row=0;row<2;row++)
    {
        for(col=0;col<3;col++)
        {
            printf("\nThe Number at [%d][%d] is %d", row, col,
arr[row][col];
        }
    }
}
```

A sample output is given below :

```
Enter a Number at [0][0] : 10
Enter a Number at [0][1] : 100
Enter a Number at [0][2] : 45
Enter a Number at [1][0] : 67
Enter a Number at [1][1] : 45
Enter a Number at [1][2] : 230
The Number at [0][0] is 10
The Number at [0][1] is 100
The Number at [0][2] is 45
The Number at [1][0] is 67
The Number at [1][1] is 45
The Number at [1][2] is 230
```

- **Two-dimensional Arrays and Strings**

As seen earlier, a string can be represented as a single-dimensional, character-type array. Each character within the string will be stored within one element of the array. This array of the string can be created using a two-dimensional character array. The left index or subscript determines the number of strings and the right index specifies the maximum length of each string. For example, the following declares an array of 25 strings, each with a maximum length of 80 characters including the null character.

```
char str_ary[25][80];
```

- **Example demonstrating the use of a Two-dimensional Array**

The following example demonstrates the use of two-dimensional arrays as strings.

Consider the problem of organizing a list of names in an alphabetical order. The following example

accepts a list of names and then arranges them in alphabetical order.

Example 6:

```
#include <stdio.h>
#include <string.h>
#include <conio.h>
void main ()
{
    int i, n = 0;
    int item;
    char x[10][12];
    char temp[12];
    clrscr();
    printf("Enter each string on a separate line \n\n");
    printf("Type 'END' when over \n\n");
    /* read in the list of strings */
    do
    {
         printf("String %d : ", n+1);
         scanf("%s", x[n]);
    } while (strcmp(x[n++], "END"));
    /*reorder the list of strings */
    n = n - 1;
    for(item=0; item<n-1; ++item)
    {
         /* find lowest of remaining strings */
         for(i=item+1; i<n; ++i)
         {
             if(strcmp (x[item], x[i]) > 0)
             {
                 /*interchange two stings */
                 strcpy(temp, x[item]);
                 strcpy(x[item], x[i]);
                 strcpy(x[i], temp);
             }
          }
    }
```

```
    /* Display the arranged list of strings */
    printf("Recorded list of strings : \n");
    for(i = 0; i < n ; ++i)
    {
        printf("\nString %d is %s", i+1, x[i]);
    }
}
```

The program accepts strings till the user types END. When END is typed, the program sorts the list of strings and prints it out in sorted order. The program checks two consecutive elements. If they are not in proper order, then they are interchanged. The comparison of two strings is done with the help of the `strcmp()` function whereas the interchanging is done with the `strcpy()` function.

A sample output of the above example is as follows:

```
Enter each string on a separate line
Type 'END' when over
String 1 : has
String 2 : seen
String 3 : alice
String 4 : wonderland
String 5 : END
Record list of strings:
String 1 is alice
String 2 is has
String 3 is seen
String 4 is wonderland
```

Summary

- An array is a collection of data elements of the same type that are referenced by a common name.
- Each element of the array has the same data type, same storage class and common characteristics.
- Each element is stored in successive locations of the main store. The data elements are known as the members of the array.
- The dimension of an array is determined by the number of indices needed to uniquely identify each element.
- Arrays can have the data type as int, char, float, or double.
- The element of an array can be referred using a variable or an integer expression.
- Automatic arrays cannot be initialized, unless each element is given a value separately.
- The extern and static arrays can be initialized when they are defined.
- A two-dimensional array can be thought of as an array of single-dimensional arrays.

Concepts

Check Your Progress

1. An _____ is a collection of data elements of the same type that are referred by a common name.

   A. Loop B. Array

   C. Structure D. None of the above

2. Each member of an array is identified by the unique _____ or _____ assigned to it.

   A. Index, Subscript B. Bound, Index

   C. None of the above

3. An array name and a variable name can be the same. (T/F)

4. Each element of an array cannot be used where a variable is allowed or required. **(T/F)**

5. Two arrays, even if they are of the same type and size, cannot be tested for _____.

   A. Condition B. Negation

   C. Equality D. None of the above

6. String can be defined as a character type array, which is terminated by a _____ character.

   A. semicolon B. comma

   C. NULL D. None of the above

7. Arrays can have more than one dimension. **(T/F)**

8. The comparison of two strings is done with the help of _____ whereas the interchanging is done by _____.

   A. `strcmp`,`strcpy` B. `strcat`,`strcpy`

   C. `strlen`,`strcat` D. None of the above

Try It Yourself

1. Write a program to arrange the following names in alphabetical order.

   George

   Albert

   Tina

   Xavier

   Roger

   Tim

   William

2. Write a program to count the number of vowels in a line of text.

3. Write a program that accepts the following numbers in an array and reverses the array.

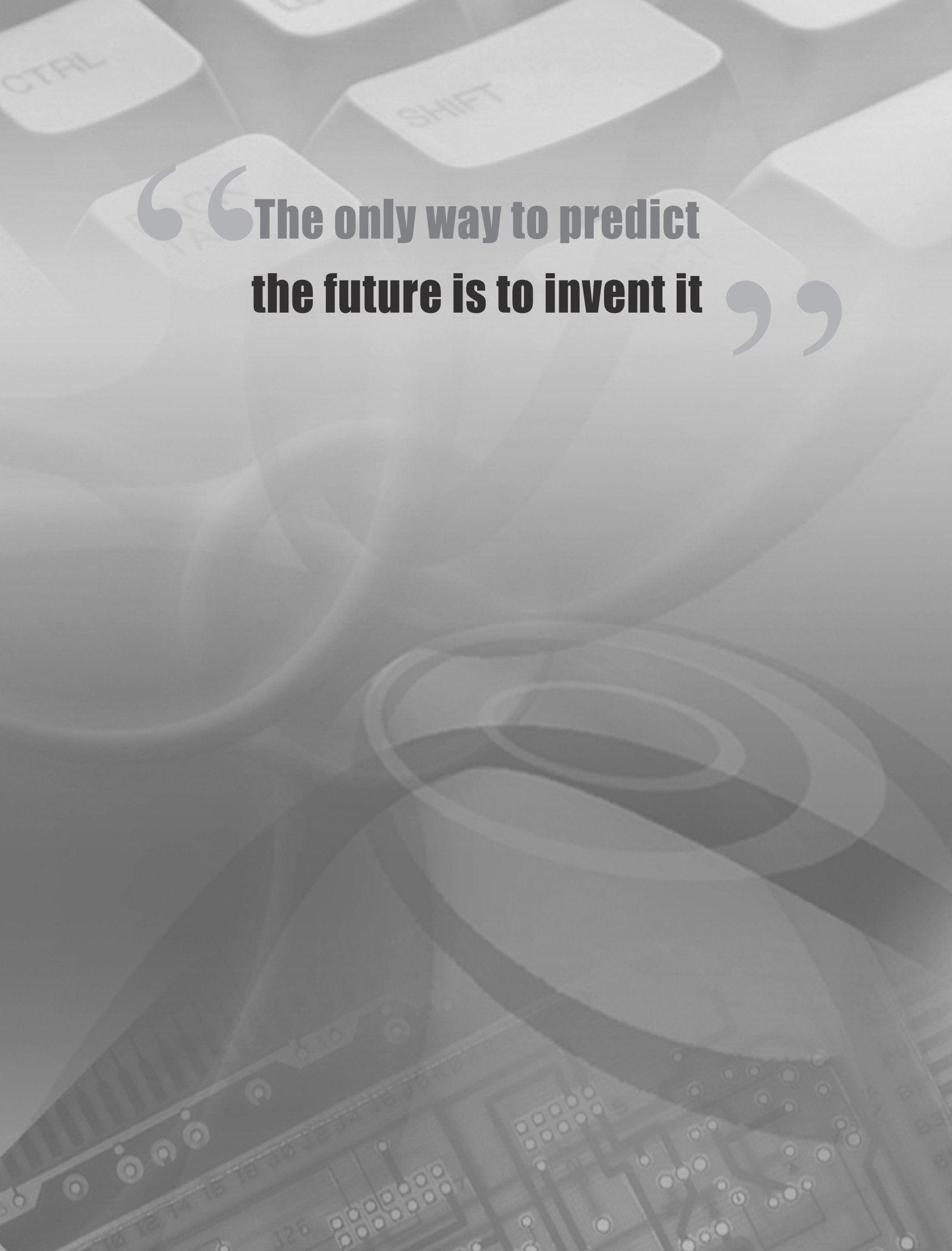
   34

   45

   56

   67

   89



“The only way to predict
the future is to invent it”

Session
12 Arrays (Lab)

Lab Guide

Objectives

At the end of this session, you will be able to:

- *Use a Single Dimensional Array*
- *Use a Two Dimensional Array*

Introduction

The steps given in this session are detailed, comprehensive and carefully thought through. This has been done so that the learning objectives are met and the understanding of the tool is complete. Please follow the steps carefully.

Part I – For the first 1 Hour and 30 Minutes:

12.1 Arrays

Arrays can be classified into two types based on dimensions: Single dimensional or Multi dimensional. In this session, let us focus on how to create and use arrays.

12.1.1 Sorting a single dimensional array

A single dimensional array can be used to store a single set of values of same data type. Consider a set of student marks in a particular subject. Let us sort these marks in descending order.

The steps to sort a single dimensional array in descending order are:

1. Accept the total number of marks to be entered.

To do this, a variable has to be declared and the value for the same has to be accepted. The code will be:

```
int n;
printf("\n Enter the total number of marks to be entered : ");
scanf("%d",&n);
```

2. Accept the set of marks.

To accept a set of values for an array the array should be declared. The code for the same is,

```
int num[100];
```

The number of elements in the array is determined by the value entered for the variable **n**. The **n** elements of the array should be initialized with values. To accept **n** values, a `for` loop is required. An integer variable needs to be declared as the subscript of the array. This variable helps in accessing individual elements of the array. The values of the array elements are then initialized by accepting values from the user. The code for the same is:

```
int l;
for(l=0;l<n;l++)
{
    printf("\n Enter the marks of student %d : ", l+1);
    scanf("%d",&num[l]);
}
```

As subscripts of the array always start from 0 and we need to initialize 1 to 0. Each time the `for` loop is executed, an integer value is assigned to the element of the array.

3. Make a copy of the array.

Before sorting the array, it is always safer to preserve the original array. Hence another array can be declared and the elements of the first array can be copied into this new array. The following lines of code are used to do this:

```
int desnum[100],k;

for(k=0;k<n;k++)
    desnum[k] = num[k];
```

4. Sort the array in descending order.

To sort an array, the elements in the array need to be compared with each other. The best way of sorting an array, in descending order, is to pick the highest value of the array and exchange it with the first element. Once this is done, the second highest element from the remaining elements can be exchanged with the second element of the array. While doing so, the first element of the array can be ignored as it is the highest of all. Similarly, the elements of the array can be eliminated one by one till the nth highest element is found. In case the array needs to be sorted in ascending order the highest value can be exchanged with the last element of the array.

Let us consider a numerical example to understand the same. Figure 12.1 represents an array of integers that need to be sorted.

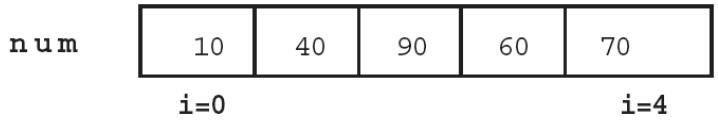


Figure 12.1: Array num with subscript i (5 elements)

To sort this array in descending order,

a. We need to find the first highest element and place it in the first position. This can be referred to as the first pass. To get the highest element to the first position, we need to compare the first element with the rest of the elements. In case the element being compared is greater than the first element then the two elements need to be exchanged.

To start with, in the first pass, the element in the first place is compared with the element in the second place. Figure 12.2 represents the exchange of the element in the first place.

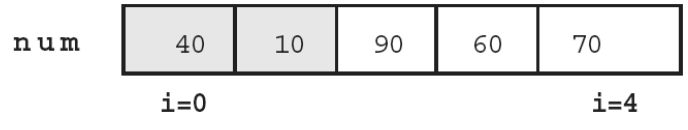


Figure 12.2: Swapping the first element with the second element

The first element is then compared with the third element. Figure 12.3 represents the exchange of the first and the third elements.

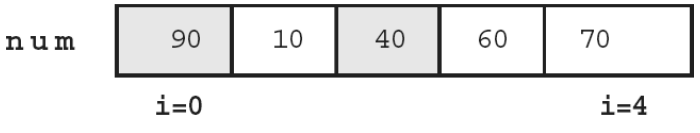


Figure 12.3: Swapping the first element with the third element

This process is repeated until the first element is compared with the last element of the array. The resultant array after the first pass will be as shown in figure 12.4.

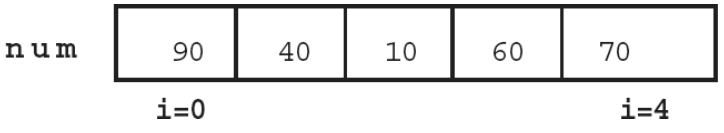


Figure 12.4: Array after the first pass

b. Leaving the first element intact, we need to find the second highest element and swap it with the second element of the array. Figure 12.5 represents the array after doing so.

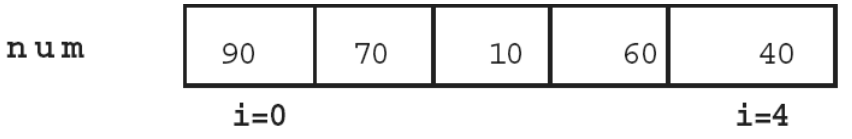


Figure 12.5: Array after second pass

c. The third element has to be swapped with the third highest element of the array. Figure 12.6 represents the array after swapping the third highest element.

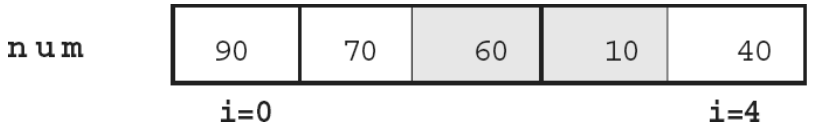


Figure 12.6: Array after third pass

d. The fourth element has to be swapped with the fourth highest element of the array. Figure 12.7 represents the array after swapping the fourth highest element.

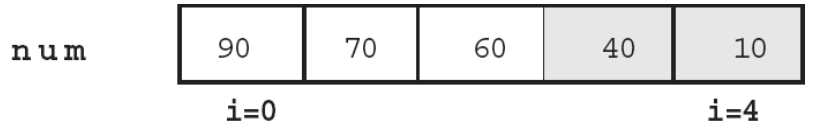


Figure 12.7: Array after fourth pass

e. Figure 12.7 also represents the sorted array.

To do this programmatically, we need two loops one to find the highest element in an array and another to repeat the process **n** times. To be more specific the process has to be repeated **n-1** times `for` an n element array because the last element will not have any other element to be compared with. Hence we declare two variables named **i** and **j** to work through two `for` loops. The `for` loop with index **i** is used to repeat the process of locating highest element in the remaining portion of the array. The `for` loop with index **j** is used to find the **i**th highest element of the array between the **i+1**th and the last element of the array. Thus the **i**th highest element in the remaining part of the array will be pushed to the **i**th position.

The code for declaring the subscripts and looping through the array **n-1** times with **i** as index is,

```
int i,j;
for(i=0;i<n-1;i++)
{
```

The code for looping from the **i+1th** element to the nth element of the array is,

```
    for(j=i+1;j<n;j++)
    {
```

To swap two elements in an array we need to use a temporary variable. This is because the moment the one element of the array is copied onto another element, the value in the second element is lost. To avoid the loss of value in the second element, the value needs to be preserved in a temporary variable. The code for swapping the **ith** element with the greatest element in the remaining part of the array is,

```
      if(desnum[i] < desnum[j])
      {
        temp = desnum[i];
        desnum[i] = desnum[j];
        desnum[j] = temp;
      }
   }
}
```

The for loops need to be closed and hence two extra closing parenthesis are given in the above code.

5. Display the sorted array.

The same subsrcipt **i** can be used to display the values of the array as shown in the statements below:

```
for(i=0;i<n;i++)
    printf("\n Number at [%d] is %d", i, desnum[i]);
```

Thus an array of elements can be sorted. Let us look at the complete program.

1. Invoke the editor in which you can type the C program.

2. Create a new file.

3. Type the following code :

```
void main()
{
    int n;
    int num[100];
    int l;
    int desnum[100],k;
    int i,j,temp;
    printf("\n Enter the total number of marks to be entered : ");
    scanf("%d",&n);
    clrscr();
    for(l=0;l<n;l++)
    {
        printf("\n Enter the marks of student %d : ", l+1);
        scanf("%d",&num[l]);
    }

    for(k=0;k<n;k++)
        desnum[k] = num[k];

    for(i=0;i<n-1;i++)
    {
        for(j=i+1;j<n;j++)
        {
            if(desnum[i] < desnum[j])
            {
                temp= desnum[i];
                desnum[i] = desnum[j];
                desnum[j] = temp;
            }
        }
    }
    for(i=0;i<n;i++)
        printf("\n Number at [%d] is %d", i, desnum[i]);
}
```

To see the output, follow the steps listed below:

4. **Save the file with the name arrayl.C**
5. **Compile the file, arrayl.C**
6. **Execute the program, arrayl.C**
7. **Return to the editor.**

The sample output of the above program is shown in Figure 12.8 and 12.9.

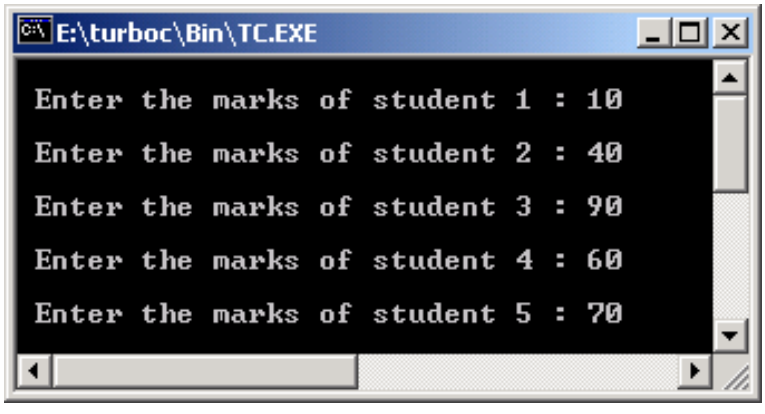


Figure 12.8: Output I of arrayl.C – Input Values

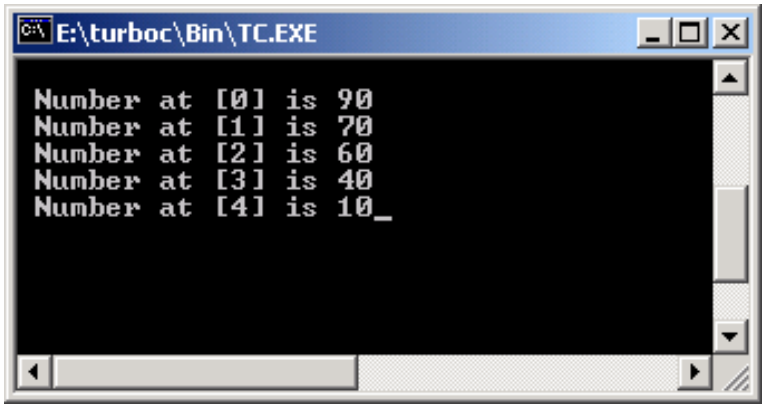


Figure 12.9 : Output II of arrayl.C – Output Values

12.1.2 Matrix addition using two dimensional arrays

Arrays can have multiple dimensions. A typical example of a two dimensional array is a matrix. A matrix is a rectangular number pattern that is made up of rows and columns. The intersection of each row and

column has a value. Figure 12.10 represents a matrix.

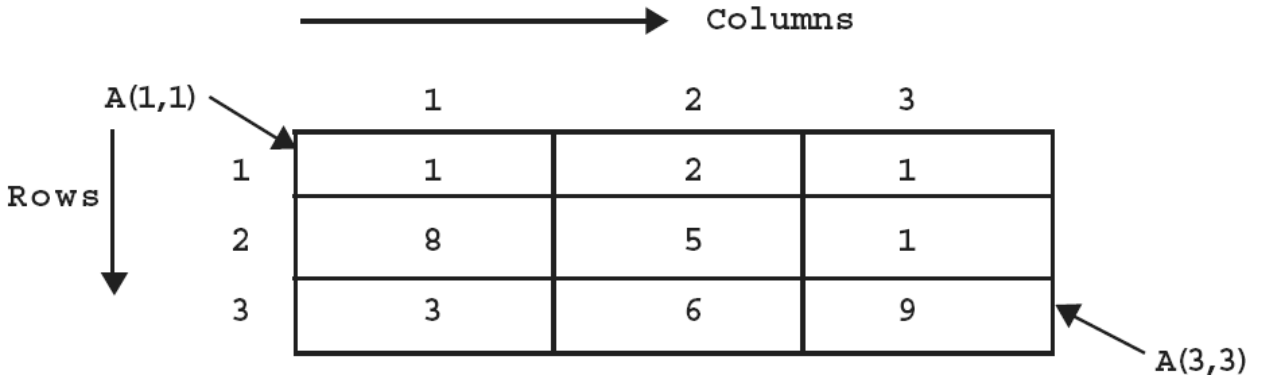


Figure 12.10 : Matrix named A

The number of rows and columns in a matrix represented as (number of rows) x (number of columns) is called the dimension of the matrix. The dimension of the matrix in Figure 12.10 is 3x3.

Let us consider the example of matrix addition for understanding the use of two dimensional arrays. Consider the following matrices A and B given in Figure 12.11.

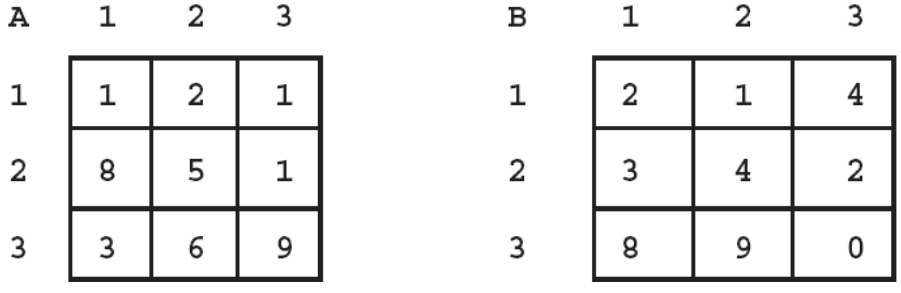


Figure 12.11: Matrix A and B

The sum of these two matrices is another matrix. This is created by adding the values from each of the corresponding rows and columns. For example, the first element `C(1,1)` in matrix `C` will be the sum of `A(1,1)` and `B(1,1)`. The second element `C(1,2)` will be the sum of `A(1,2)` and `B(1,2)` and so on. An important rule in summing matrices is that the dimension of the matrices that are being added should be same. That is, a 2x3 matrix can be added with another 2x3 matrix only. Figure 12.12 represents the matrices `A`, `B` and `C`.

A	1	2	3
1	1	2	1
2	8	5	1
3	3	6	9

B	1	2	3
1	2	1	4
2	3	4	2
3	8	9	0

C	1	2	3
1	3	3	5
2	11	9	3
3	11	15	9

Figure 12.12 : Matrix A, B and C

To do this programmatically,

1. Declare three two dimensional arrays. The code for the same is,

```
int A[10][10], B[10][10], C[10][10];
```

2. Accept the dimension of the matrices. The code is,

```
int row,col;
printf("\n Enter the dimension of the matrix : ");
scanf("%d %d",&row,&col);
```

3. Accept the values of the matrix A and B.

The values of a matrix are accepted row wise. First the values of the first row are accepted. Then the values of the second row are accepted and so on. Within a row the values of each column are accepted sequentially. Hence two for loops are necessary to accept the values of a matrix. The first for loop traverses through the rows one by one, whereas the inner for loop will traverse through the columns one by one.

The code for the same is,

```
printf("\n Enter the values of the matrix A and B : \n");
int i, j;
for(i=0;i<row;i++)
    for(j=0;j<col;j++)
    {
        print("A[%d,%d],B[%d,%d]:",row,col,row,col);
        scanf("%d %d",&A[i][j],&B[i][j]);
    }
```

4. Add the two matrices. The two matrices can be added using the following code,

```
C[i][j] = A[i][j] + B[i][j];
```

Note this line needs to be inside the inner for loop of the code given previously. Alternatively, the two for loops can be re-written to add the matrices.

5. Display the three matrices. The code for the same will be,

```
for(i=0;i<row;i++)
    for(j=0;j<col;j++)
    {
      printf("\nA[%d,%d]=%d,B[%d,%d]=%d,C[%d,%d]=%d \n",i,j,A[i][j],i,j
,B[i][j],i,j,C[i][j]);
    }
```

Let us look at the complete program.

1. Create a new file.

2. Type the following code :

```
void main()
{
    int A[10][10], B[10][10], C[10][10];
    int row,col;
    int i,j;
    printf("\n Enter the dimension of the matrix : ");
    scanf("%d %d",&row,&col);
    printf("\nEnter the values of the matrix A and B: \n");

    for(i=0;i<row+i++)
        for(j=0;j<col;j++)
        {
            print("\n A[%d,%d] , B[%d,%d] : ",i,j,i,j);
            scanf("%d %d",&A[i][j],&B[i][j]);
            C[i][j] = A[i][j] + B[i][j];
        }
```

```
    for(i=0;i<row+i++)
        for(j=0;j<col;j++)
        {
            printf("\n A[%d,%d]=%d,B[%d,%d]=d,C[%d,%d]=%d\n",i,j,A[i][j
],i,j,B[i][j],i,j,C[i][j]);
        }
}
```

3.	**Save the file with the name arrayII.C**
4.	**Compile the file, arrayII.C**
5.	**Execute the program, arrayII.C**
6.	**Return to the editor.**

The sample output of the above program will be as shown in Figure 12.13.

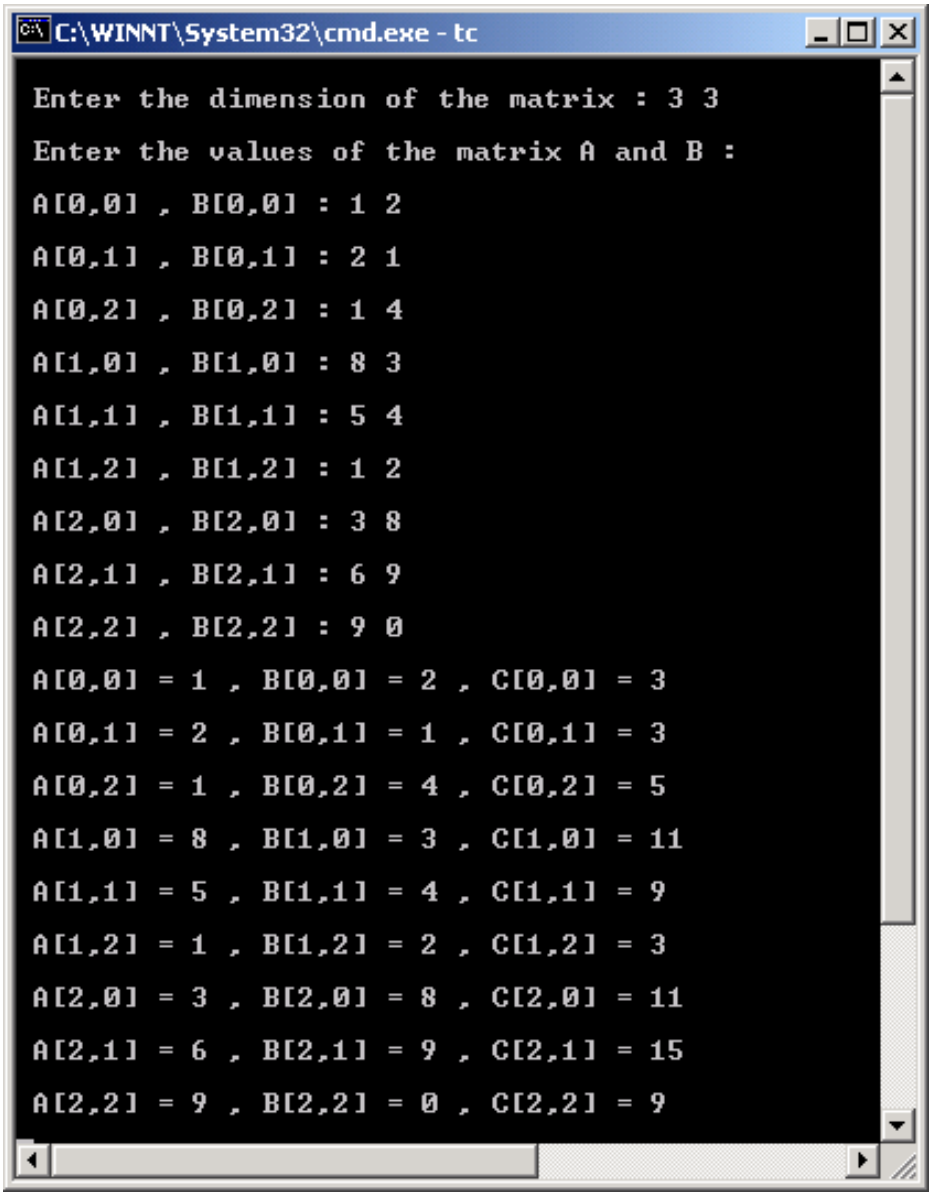


Figure 12.13 : Output I of arrayII.C – Input Values

Part II – For the next 30 Minutes :

1. Write a C program that accepts a set of numbers in an array and reverses the array.

   To do this,

   a. Declare two arrays.

   b. Accept the values of one array.

   c. Loop through the array in the reverse order to copy the values into the second array. While doing so have a different subscript to the second array which will move forward.

Try It Yourself

1. Write a C program to find the minimum and the maximum value in an array.

2. Write a C program to count the number of vowels and the number of consonants in a word.