**Chapter Three**

1. **Flow of Control**

* A running program spends all of its time executing instructions or statements in that program.
* The order in which statements in a program are executed is called ***flow***of that program.
* Programmers can control which instruction to be executed in a program, which is called ***flow contro****l*.
* This term reflects the fact that the currently executing statement has the control of the CPU, which when completed will be handed over (flow) to another statement.
* Flow control in a program is typically ***sequential***, from one statement to the next.
* But we can also have execution that might be divided to other paths by ***branching statements***. Or perform a block of statement repeatedly until a condition fails by ***Repetition*** or ***looping***.
* Depending upon requirement of the problem, it is often required to alter the normal sequence of execution in the program. This means that we may desire to selectively or repetitively execute a program statement.
* Flow control is an important concept in programming because it will give all the power to the programmer to decide what to do to execute during a run and what is not, therefore, affecting the overall outcome of the program.

# Sequential Statements

* Such kinds of statements are instruction in a program which will be executed one after the other in the order scripted in the program. In sequential statements, the order will be determined during program development and cannot be changed.
* A number of C++ structure known as control structures are available for the flow of processing.

# 3.1 Selection Statements/Conditional statements

* Selection statements are statements in a program where there are points at which the program will decide at runtime whether some part of the code should or should not be executed.
* They are mainly used for making decisions. Depending on the value of an expression decision can be made.
* There are two types of selection statements in C++, which are the “***if statement***” and the “***switch statement***”

## 3.1.1 The if Statement

* + It is sometimes desirable to make the execution of a statement dependent upon a *condition being satisfied*.
  + The different forms of the ‘If” statement will be used to decide whether to execute part of the program based on a condition which will be tested either for TRUE or FALSE result.
  + The different forms of the “If” statement are:
    - The simple if statement
    - The if else statement
    - The if else if statement

***The simple if statement***

* + The simple if statement will decide only one part of the program to be executed if the condition is satisfied or ignored if the condition fails.
  + The General Syntax is:

*If (expression)*

*Statements;*

* + In any “*if*” statement, first the “*expression”* will be evaluated and if the outcome is non zero (which means TRUE), then the “*statements”* is executed. Otherwise, nothing happens (the statement will not be executed) and execution resumes to the line immediately after the “if” block.
  + To make multiple statements dependent on the same condition we can use a compound statement, which will be implemented by embracing the group of instructions within the left “{“and right “}” French bracket.
  + The following diagram shows the execution of if statement.

**Expression**

**False**

**True**

Body of if

E.g. if(age>18)

cout<<”you are an adult”;

E.g.:

if(balance > 0)

{

interest = balance \* creditRate;

balance += interest;

}

* + Most of the time “*expression”* will have relational expressions testing whether something is equal, greater, less, or different from something else.
  + It should be noted that the output of a relational expression is either True (represented by anything different from zero) or False (represented by Zero).
  + Thus any expression, whose final result is either zero or none zero can be used in ”*expression*”

E.g.:

int x;

cin>>x;

if(x)

cout<<”you are an adult”;

* + In the above example, the value of variable x will be an input from the user. The “if” statement will be true if the user gives anything different from zero, which means the message “***you are an adult***” will be displayed on the screen. If the user gives zero value for x, which will be interpreted as False, nothing will be done as the if statement is not satisfied.
  + Thus, expression can be:
    - Relational expression,
    - A variable,
    - A literal constant, or
    - Assignment operation, as the final result is whatever we have at the right hand side and the one at the left hand side is a variable.

***The if else statement***

* + Another form of the “if” is the “if …else” statement.
  + The “if else” statement allows us to specify two alternative statements:
    - One which will be executed if a condition is *satisfied* and
    - Another which will be executed if the condition is *not satisfied*.
  + The General Syntax is:

*if (expression)*

*statements1;*

*else*

*statements2;*

* + First “*expression*” is evaluated and if the outcome is none zero (true), then “*statements1*” will be executed. Otherwise, which means the “*expression”* is false “*statements2”* will be executed.
  + The following diagram shows the execution of if else statement.

Expression

**False**

**True**

Body of else

Body of if

E.g.:

if(balance > 0)

{

interest = balance \* creditRate;

balance += interest;

}

else

{

interest = balance \* debitRate;

balance += interest;

}

**E.g.1:**

if(age>18)

cout<<”you are an adult”;

else

cout<<”You are a kid”;

**E.g.2:**

int x;

cout<<”Enter a number: “;

cin>>x;

if(x%2==0)

cout<<”The Number is Even”;

else

cout<<”The Number is Odd”;

**E.g.3:**

int x;

cout<<”Enter a number: “;

cin>>x;

if(x%2)

cout<<”The Number is Odd”;

else

cout<<”The Number is Even”;

* + The above three examples illustrate the “if else” statement.
  + The last two examples will do the same thing except that the expression in the Eg3 is changed from relational to arithmetic. We know that the result from the modulus operator is either 0 or one if the divider is 2. Thus the final result of the expression in Eg3 will either be 1 for odd numbers or be 0 for even numbers.
  + ‘if’ statements may be nested by having an if statement appear inside another if statement. For instance:

if(callHour > 3){

if(callDuration <= 5)

charge = callDuration \* tarif1;

else

charge = 5 \* tarif1 + (callDuration - 5) \* tarif2;

}

else

charge = flatFee;

**E.g:**

#include<iostream.h>

#include<conio.h>

int main()

{

int testScore;

cout<<”\nEnter your test score:”;

cin>>testScore;

if(testScore < 70)

cout<<”\n You didn’t pass:”; // then block

else

cout<<”\n You did pass”; //else block

return 0;

}

* + the above sample program can be diagrammatically expressed as follows:

**testScore < 70?**

cout<<”you did pass”;

cout<<”you didn’t pass”;

**False**

**True**

***The “if else if” statement***

* + The third form of the “if” statement is the “if …else if” statement.
  + The “if else if” statement allows us to specify more than two alternative statements each will be executed based on testing one or more conditions.
  + The General Syntax is:

*if (expression1)*

*statements1;*

*else if(expression2)*

*statements2;*

**.**

**.**

**.**

*else if(expressionN)*

*statementsN;*

*else*

*statements*

* + First “*expression1*” is evaluated and if the outcome is none zero (true), then “*statements1*” will be executed. Otherwise, which means the “*expression1”* then “*expression2”* will be evaluated: if the out put of expression2 is True then “*statements2”* will be executed otherwise the next “*expression”* in the else if statement will be executed and so on.
  + If all the expressions in the “if” and “else if” statements are False then “*statements”* under else will be executed.
  + The “if…else” and “if…else if” statements are said to be exclusive selection as if one of the condition (expression) is satisfied only instructions in its block will be executed and the rest will be ignored.

E.g.:

if(score >= 90)

cout<< “\n your grade is A”;

else if(score >= 80)

cout<< “\n your grade is B”;

else if(score >= 70)

cout<< “\n your grade is C”;

else if(score >= 60)

cout<< “\n your grade is D”;

else

cout<< “\n your grade is F”;

* + In the above example, only one of the five cout statements will be executed and the rest will be ignored. But until one of the conditions is satisfied or the else part is reached, the expressions will be tested or evaluated.

***Nesting If statements within another if statement***

* + One or more if statements can be nested with in another if statement.
  + The nesting will be used to test multiple conditions to perform a task.
  + It is always recommended to indent nested if statements to enhance readability of a program
  + The General Syntax might be:

*if (expression1)*

*{*

*if (expression2)*

*statementsN;*

*else*

*statementsM;*

*}*

*else*

*{*

*if (expression3)*

*statementsR;*

*else*

*statementsT;*

*}*

* + StatementsN will be executed if and only if “*expression1*” and “*expression2*” are evaluated and if the outcome of both is none zero (TRUE).
  + StatementsM will be executed if and only if “*expression1*” is TRUE and “*expression2*” is FALSE.
  + StatementsR will be executed if and only if “*expression1*” is FALSE and “*expression3*” is TRUE.
  + StatementsT will be executed if and only if “*expression1*” is FLASE and “*expression2*” is FALSE.
  + Lets have an example on nested if statements

#...

int main()

{

int testScore, age;

cout<<”\n Enter your test score :”;

cin>>testScore;

cout<<“\n enter your age:”;

cin>>age;

if(testScore >= 70)

{

if(age < 10)

cout<<“\n You did a great job”;

else

cout<<“\n You did pass”;

}

else

cout<<“\n You did not pass”;

getch();

return 0;

}

* + The above program can be expressed diagrammatically as follows.

testScore >=70?

cout<</;you didn’t pass”;

age < 10?

cout<<”you did pass”;

cout<<you did a great job”;

testScore >=70?

cout<</;you didn’t pass”;

age < 10?

cout<<”you did pass”;

cout<<you did a great job”;

testScore >=70?

cout<</;you didn’t pass”;

age < 10?

cout<<”you did pass”;

cout<<you did a great job”;

### 3.1.2 The Switch Statement

* Another C++ statement that implements a selection control flow is the switch statement (*multiple-choice statement*).
* The switch statement provides a way of choosing between a set of alternatives based on the value of an expression. The general form of the switch statement is:
* The switch statement has four components:
  + - ***Switch***
    - ***Case***
    - ***Default***
    - ***Break***
* Where Default and Break are Optional
* The General Syntax might be:

*switch(expression)*

*{*

*case constant1:*

*statements;*

***.***

***.***

*case constant n:*

*statements;*

*default:*

*statements;*

*}*

* Expression is called the ***switch tag*** and the constants preceding each case are called the ***case labels***.
* The output of “*expression*” should always be a constant value.
* First expression is evaluated, and the outcome, which is a constant value, will compared to each of the numeric constants in the case labels, in the order they appear, until a match is found.
* Note, however, that the evaluation of the ***switch tag*** with the ***case labels*** is only for ***equality.***
* The statements following the matching case are then executed. Note the plural: each case may be followed by zero or more statements (not just one statement).
* After one case is satisfied, execution continues until either a ***break*** statement is encountered or all intervening statements are executed, which means until the execution reaches the right French bracket of the switch statement.
* The final default case is optional and is exercised if none of the earlier cases provide a match. This means that, if the value of the “*expression*” is not equal to any of the case labels, then the statements under ***default*** will be executed.
* Now let us see the effect of including a break statement in the switch statement.

Scenario One Scenario Two

switch (N){

case 1: x=10;

case 2: x=20;

case 3: x=30;

}

Even if N is 1 or 2 x will have 30

switch(N){

case 1: x=10; break;

case 2: x=20; break;

case 3: x=30; break;

}

X will have 10, 20 or 30 based on the value of N

x = 10

x =20

x = 30

x = 20

break

x = 10

x = 30

break

break

**True True**

**False**

**False**

**True True**

**False** false

**True** **True**

**False** **False**

* The break terminates the switch statement by jumping to the very end of it.
* There are, however, situations in which it makes sense to have a case without a break. For instance:

E.g.: switch(operator)

{

case ‘+’: result = operand1 + operand2;

break;

case ‘-’ : result = operand1 – operand2;

break;

case ‘x’:

case ‘\*’: result = operand1 \* operand2;

break;

case ‘/’: result = operand1 / operand2;

break;

default: cout<<“ unknown operator:”<<operator<<“\n”;

}

* Because case ‘x’ has no break statement (in fact no statement at all!), when this case satisfied, execution proceeds to the statements of the next case and the multiplication is performed.
* Switch evaluates expression and compares the result to each of the case values.
* Relational and Boolean operators can be used in switch tag if and only if the expected output is either 0 to represent False or 1 to represent True as that is the only possible output from such operators.

# 3.2 Repetition/Loop Statements

* Repetition statements control a block of code to be executed repeatedly for a fixed number of times or until a certain condition fails.
* Loop causes a section of our program to be repeated a certain number of times. The repetition continues up to when a condition is true.
* There are three C++ repetition statements:

1. The ***For*** Statement or loop
2. The ***While*** statement or loop
3. The ***do…while*** statement or loop

### 3.2.1 The for statement / loop

* + The “for” statement (also called loop) is used to repeatedly execute a block of instructions until a specific condition fails.
  + The General Syntax is:

*for(expression1 ; expression2 ; expression3)*

*statements;*

* + The for loop has three expressions:
    - ***expression1***: is one or more statements that will be executed only once and before the looping starts. ***Expression2***: is the part that decides whether to proceed with executing the instructions in the loop or to stop. Expression2 will be evaluated each time before the loop continues. The output of expression2 should be either non zero (to proceed with the loop) or zero (to stop the loop) to represent true and false output respectively.
    - ***Expression3***: is one or more statements that will be executed after each iteration.
  + Thus, first ***expression1*** is evaluated and then each time the loop is executed, ***expression2*** is evaluated. If the outcome of ***expression2*** is non zero then ***statements*** is executed and ***expression3*** is evaluated. Otherwise, the loop is terminated.
  + In most programs, the “for loop” will be used for such expressions where ***expression1*** is ***initialization***, ***expression2*** is ***condition*** and ***expression3*** is either ***increment*** or ***decrement***.
  + The general format can be expressed as follows for the sake of clarity:

*for(initialization ; condition ; increase/decrease)*

*statement;*

* Steps of execution of the for loop:

1. *Initialization is executed. (will be executed only once)*
2. *Condition is checked, if it is true the loop continues, otherwise the loop finishes and statement is skipped.*
3. *Statement is executed.*
4. *Finally, whatever is specified in the increase or decrease field is executed and the loop gets back to step two.*

E.g. guess the output of the following code:

int main()

{

for(int i=10;i>0;i--)

{

cout<<n<<“,”;

}

cout<< “FIRE!”;

return 0;

}

* + Even though it is not recommended, *expression1*, *expression2* and *expression3* can be optional or can be ignored. This means that they can take NULL statement.
  + But making ***expression2*** null means that the loop will not terminate. In such cases one can include an “if” statement inside the “for” loop which will test a condition and break out from the loop using the ***break*** statement.
  + While making one or more of the three expressions null, the semi colons CAN NOT be ignored.

E.g.:

for (;n<10;) //if we want neither initialization nor increase/decrease

for (;n<10;n++) //if no initialization is needed.

for ( ; ; ) //is an infinite loop unless and otherwise there is if statement inside the loop.

* + It is declared above that expression1 and expression3 can be one or more statements. The composite statements should be separated by a comma. This means, optionally, using the comma operator (,) we can specify more than one instruction in any of the two fields included in a “for” loop.

E.g.

for(n=0,i=100;n!=i; n++,i--){

//whatever here

}

* + In the above example, n=0 and i=100 will be part of expression1 and will be executed only once before the loop starts. In addition, n++ and i—will be part of expression3 and will be executed after each looping/iteration.

Eg:1

//the following for statement adds the numbers between 0 and n

*int Sum=0;*

*for(int i=0; i<=n;i++)*

*Sum=Sum+i;*

Eg:2

//the following for statement adds the even numbers between 0 and n

*int Sum=0;*

*for(int i=0; i<=n;)*

*{*

*Sum=Sum+i;*

*i+=2;*

*}*

Eg:3

//the following for statement adds the even numbers between 0 and n

//where all the three expressions are null.

*int Sum=0;*

*int i=0;*

*for( ; ; )*

{

*If(i<=n)*

*break;*

*else*

{

*Sum=Sum+i;*

*i++;*

}

}

* + In the above example, the initialization is at the top before the looping starts, the condition is put in if statement before the instructions are executed and the increment is found immediately after the statements to be executed in the loop.

**NB:** even though there is the option of making all the three expressions null in a “for” loop, it is not recommended to make the condition to take null statement.

### 3.2.2 The while statement

* + The while statement (also called while loop) provides a way of repeating a statement or a block as long as a condition holds / is true.
  + The general form of the while loop is:

*while(expression)*

*statements;*

* + First *expression* (called the *loop condition*) is evaluated. If the outcome is non zero then statement (called the *loop body*) is executed and the whole process is repeated. Otherwise, the loop is terminated.
  + Suppose that we wish to calculate the sum of all numbers from 1 to some integer value n. This can be expressed as :

E.g.1:// adds the numbers between 0 and any given number n

i=1;

sum = 0;

while(i <= n)

sum += i++;

E.g.2://adds the numbers between 0 and 100

number=1;

sum=0;

while(number <= 100)

{

sum += number;

number++;

}

E.g.3:

/\*the following while loop will request the user to enter his/her age which should be between 0 and 130. If the user enters a value which is not in the range, the while loop test the value and request the age again until the user enters a valid age.\*/

cout<<“\n Enter your age [between 0 and 130]:”;

cin>>age;

while(age < 0 || age > 130)

{

cout<<“\n invalid age. Plz try again!”;

cin>>age;

}

**3.2.3 Do…while loop**

* + The do statement (also called the do while loop) is similar to the while statement, except that its body is executed first and then the loop condition is examined.
  + In do…while loop, we are sure that the body of the loop will be executed at least once. Then the condition will be tested.
  + The general form is:

*do*

*{*

*statement;*

*}*

*while(expression);*

* + First statement is executed and then expression is evaluated. If the outcome of the expression is nonzero, then the whole process is repeated. Otherwise the loop is terminated.

E.g.:

//our previous example (Eg3) in the while loop might be changed as:

*age=-1;*

*do*

*{*

*cout<<“\n enter your valid age [between 0 and 130]:”;*

*cin>>age;*

*}*

*while(age < 0 || age > 130);*

* E.g. what do you think is the outcome of the following code:

*unsigned long n;*

*do*

*{*

*cout<<“\n enter number (0 to end):”;*

*cin>>n;*

*cout<<“\n you entered:”<<n;*

*}*

*while(n != 0);*

### Pitfalls in writing repetition statements

* + There are some pitfalls worth mentioning in repetition statements. These pit falls are the most common programming errors committed by programmers
* ***Infinite loop***: no matter what you do with the while loop (and other repetition statements), make sure that the loop will *eventually terminates*.

E.g.1:

//Do you know why the following is an infinite loop?

int product = 0;

while(product < 50)

product \*= 5;

E.g.2:

//Do you know why the following is an infinite loop?

int counter = 1;

while(counter != 10)

counter += 2;

* + In the first example, since ***product*** is initialized with zero, the expression “product\*=5” will always give us zero which will always be less than 50.
  + In the second example, the variable counter is initialized to 1 and increment is 2, counter will never be equal to 10 as the counter only assumes odd values. In theory, this while loop is an infinite loop, but in practice, this loop eventually terminates because of an *overflow error* as counter is an integer it will have a maximum limit.
* ***Off-By-One Bugs (OBOB)***: another thing for which you have to watch out in writing a loop is the so called *Off-By-One Bugs or errors*. Suppose we want to execute the loop body 10 times. Does the following code work?

E.g.:1

count = 1;

while(count < 10)

{

count++;

}

No, the loop body is executed nine times. How about the following?

E.g.:2

count = 0;

while(count <= 10)

{

…

count++;

}

No this time the loop body is executed eleven times. The correct is

E.g.:3

count = 0;

while(count < 10)

{

…

count++;

}

OR

count = 1;

while(count <= 10)

{

…

count++;

}

## The continue and break statements

### The continue statement

* The continue statement terminates the current iteration of a loop and instead jumps to the next iteration.
* It is an error to use the continue statement outside a loop.
* In while and do while loops, the next iteration commences from the loop condition.
* In a “for” loop, the next iteration commences from the loop’s third expression.

E.g.:

for(int n=10;n>0;n--)

{ if(n==5)

continue; //causes a jump to n—

cout<<n<< “,”;

}

* When the continue statement appears inside nested loops, it applies to the loop immediately enclosing it, and not to the outer loops. For example, in the following set of nested loops, the continue statement applies to the “for” loop, and not to the “while” loop.

E.g.:

while(more)

{

for(i=0;i<n;i++)

{ cin>>num;

if(num<0)

continue; //causes a jump to : i++

}

}

### The break statement

* A break statement may appear inside a loop (***while***, ***do***, or ***for***) or a switch statement. It causes a jump out of these constructs, and hence terminates them.
* Like the continue statement, a break statement only applies to the “loop” or “switch” immediately enclosing it. It is an error to use the break statement outside a loop or a switch statement.

E.g.:

for(n=10;n>0;n--)

{ cout<<n<< “,”;

if(n = = 3)

{ cout<< “count down aborted!!”;

break;

}

}

**Worksheet 3**

**Make sure to use looping whenever applicable.**

1. Write for, do-while, and while statements to compute the following sums and products.
2. 1+2+3+…+100
3. 5+10+15+…+50
4. 1+1/2+1/3+1/4+…1/15
5. 1\*2\*3\*…\*20
6. write an application to print out the numbers 10 through 49 in the following manner

10 11 12 13 14 15 16 17 18 19

20 21 22 23 24 25 26 27 28 29

30 31 32 33 34 35 36 37 38 39

40 41 42 43 44 45 46 47 48 49

1. A prime number is an integer greater than one and divisible only by itself and one. The first seven prime numbers are 2, 3, 5, 7, 11, 13, and 17. Write a program that displays all the prime numbers between 1 and 100.
2. Write a C++ program that counts the number of digits in an integer number. For example; 23,498 has five digits.
3. Write a C++ application that can compute the letter grade of a student after accepting the student’s mid and final mark. The program should only accept mid result [0-40] and final [0- 60]. If the data entered violates this rule, the program should display that the user should enter the mark in the specified range. The program is also expected to run until the user refuses to continue.
4. Write a C++ program that accepts a positive number from the user and displays the factorial of that number. Use for loops to find the factorial of the number.
5. Write a C++ code that computes the sum of the following series.

Sum = 1! + 2! + 3! + 4! + …n!

The program should accept the number from the user.

1. Using the ASCII table numbers, write a program to print the following output, using a nested for loop. (Hint: the outer loop should loop from 1 to 5, and the inner loop’s start variable should be 65, the value of ASCII “A”).

A

AB

ABC

ABCD

ABCDE

1. Write a C++ program that displays the following output using their ASCII values.

a

bc

def

gehi

jklmn

opqrst

1. Write a C++ program that will print the following shapes.

**A.**

**\***

**\*\***

**\*\*\***

**\*\*\*\***

**\*\*\*\*\***

**B.**

**\*\*\*\*\***

**\*\*\*\***

**\*\*\***

**\*\***

**\***

**C.**

**\***

**\*\*\***

**\*\*\*\*\***

**\*\*\*\*\*\*\***

**\*\*\*\*\*\*\*\*\***

**D.**

**\***

**\*\*\***

**\*\*\*\*\***

**\*\*\***

**\***

1. Write a weather-calculator program that asks for a list of the previous 10 days’ temperatures, computes the average, and prints the results. You have to compute the total as the input occurs, then divide that total by 10 to find the average. Use a while loop for the 10 repetitions.
2. Write a C++ program that accepts marks of five students and then displays their average. The program should not accept mark which is less than 0 and mark greater than 100.
3. Develop a calculator program that computes and displays the result of a single requested operation.

E.g. if the input is

15 \* 20, then the program should display 15 \* 20 equals 300

If the operator is not legal, as in the following example

24 ~ 25 then the program displays ~ is unrecognized operator

As a final example, if the denominator for a division is 0, as in the following input: 23 / 0 then the program should display the following:

23 / 0 can’t be computed: denominator is 0.

1. Use either a switch or an if-else statement and display whether a vowel or a consonant character is entered by the user. The program should work for both lower case and upper case letters.
2. Write a C++ code to display only even numbers found between 0 and 20.
3. Write a C++ application that extracts a day, month and year and determine whether the date is valid. If the program is given a valid date, an appropriate message is displayed. If instead the program is given an invalid date, an explanatory message is given. Note: to recognize whether the date is valid, we must be able to determine whether the year is a leap year or not.

An example of the expected input/output behavior for a valid date follows

Please enter a date (dd mm yyyy) : 30 4 2006

30/4/2006 is a valid date

Please enter a date (dd mm yyyy) : 1 13 2006

Invalid month: 13

Please enter a date (dd mm yyyy) : 29 2 1899

Invalid day of month 29

If the year is a leap year, then February will have total of 29 days. Otherwise, it will have 28 days. If the year is not a century year and is evenly divisible by 4, then the year is a leap year. If the year is a century year (years whose last digits are 00) and is evenly divisible by 400, then the year is a leap year.