Chapter Three: Flow of Control

3.1 Introduction

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***. 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.

3.2 Boolean values

A boolean type is an integral type whose variables can have only two values: **false** and **true**. These values are stored as the integers **0** and **1**. The boolean type in Standard C++ is named **bool**.

Example, Boolean Variables

```
int main()
{ // prints the value of a boolean variable:
        bool flag=false;
        cout << "flag = " << flag << endl;
        flag = true;
        cout << "flag = " << flag << endl;
}
```

A ***boolean expression*** is a condition that is either true or false.

Example:-

```
if (n%d) cout << "n is not a multiple of d"; // (n%d) is a Boolean expression.
```

The output statement will execute precisely when n%d is not zero, and that happens precisely when d does not divide n evenly, because n%d is the remainder from the integer division.

3.3 Conditional statements

3.3.1 The *If* statement/selection structure

It is sometimes desirable to make the execution of a statement dependent upon a condition being satisfied. The if statement provides a way of expressing this, the general form of which is:

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

The pseudocode statement:-

If student's grade is greater than or equal to 60

Print "Passed"

determines whether the condition "student ' s grade is greater than or equal to 60" is true or f a l s e . If the condition is true, then "Passed" is printed and the next pseudocode statement in order i s "performed" (remember that pseudocode is not a real programming language). If the condition is false, the print statement is ignored and the next pseudocode statement in order i s performed. Note that the second line of this selection structure is indented. Such indentation is optional, but it is highly recommended because it emphasizes the inherent structure of structured program s. When you convert your pseudocode into C++ code, the C++ compiler ignores whitespace characters (like blanks, tabs and newlines) used for indentation and vertical spacing.

The preceding pseudocode *If* statement can be wri tten in C++ as

```
if ( grade > = 6 0 )
    cout « " Passed " ;
```

3.3.2 The *If/else* statement/selection structure

The if/else selection structure allows the programmer to specify an action to perform when the condition is true and a different action to perform when the condition is false. For example, the pseudocode statement

If student's grade is greater than or equal to 60

Print "Passed "

else

Print " Failed "

prints ***Passed*** if the student ' s grade is greater than or equal to 60, but prints ***Failed*** if the student' s grade is less than 60. I n either case, after printing occurs, the next pseudocode statement in sequence is "performed."

The preceding pseudocode if/else structure can be written in C++ as:

if (grade > = 6 0)

cout « " Passed " ;

else

cout « "Failed" ;

Note that the body of the else is also indented. Whatever indentation convention you choose should be applied consistently throughout your programs. It is difficult to read programs that do not obey uniform spacing conventions.

C++ provides the conditional operator (? :), which is closely related to the if/else structure. The conditional operator is C++'s only ternary operator-it takes three operands.

The above if/else statement can be replaced with the following line of code:-

cout « (grade >= 60 ? " Passed " : " Failed ") ;

3.3.3 **Nested *if/else*** statement/selection structure

Like compound statements, selection statements can be used wherever any other statement can be used. So a selection statement can be used within another selection statement. This is called nesting statements.

When **if..else** statements are nested, the compiler uses the following rule to parse the compound statement:

Match each **else** *with the last unmatched* **if**.

A frequently-used form of nested if statements involves the else part consisting of another if-else statement. For example: nested if..else statements to check if a character is a digit, upperletter, lower letter or special character:

```
int main{
    if (ch >= '0' && ch <= '9')
    kind = digit;
    else {
        if (ch >= 'A' && ch <= 'Z')
            kind = upperLetter;
        else {
```

```
                    if (ch >= 'a' && ch <= 'z')
                        kind = lowerLetter;
                    else
                        kind = special;
                }
            }
        }
```

3.3.3 The ***else if*** Construct/structure

Nested if..else statements are often used to test a sequence of parallel alternatives, where only the else clauses contain further nesting. In that case, the resulting compound statement is usually formatted by lining up the else if phrases to emphasize the parallel nature of the logic.

The above nested if else code can also be written using the ***else if*** construct as shown below:

```
int main{
        if (ch >= '0' && ch <= '9')
            kind = digit;
        else if (cha >= 'A' && ch <= 'Z')
            kind = capitalLetter;
        else if (ch >= 'a' && ch <= 'z')
            kind = smallLetter;
        else
            kind = special;

      }
```

3.3.4 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 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;
  }
```

First *expression* (called the switch **tag**) is evaluated, and the outcome is compared to each of the *constant*s (called case **labels**), in the order they appear, until a match is found. 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). Execution continues until either a break statement is encountered or all intervening statements until the end of the switch statement are executed. The final default case is optional and is exercised if none of the earlier cases provide a match.

For example, suppose we have parsed a binary arithmetic operation into its three components and stored these in variables operator, operand1, and operand2. The following switch statement performs the operation and stores the result in result.

```
switch (operator) {
    case '+':   result = operand1 + operand2;
                break;
    case '-':   result = operand1 - operand2;
                break;
    case '*':   result = operand1 * operand2;
                break;
    case '/':   result = operand1 / operand2;
                break;
    default:cout << "unknown operator: " << ch << '\n';
                break;
}
```

As illustrated by this example, it is usually necessary to include a break statement at the end of each case. 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 example, if we extend the above statement to also allow x to be used as a multiplication operator, we will have:

```
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: " << ch << '\n';
                break;
}
```

Because case 'x' has no break statement (in fact no statement at all!), when this case is satisfied, execution proceeds to the statements of the next case and the multiplication is performed.

It should be obvious that any switch statement can also be written as multiple if-else statements. The above statement, for example, may be written as:

```
if (operator == '+')
    result = operand1 + operand2;
else if (operator == '-')
    result = operand1 - operand2;
else if (operator == 'x' || operator == '*')
    result = operand1 * operand2;
else if (operator == '/')
    result = operand1 / operand2;
else
    cout << "unknown operator: " << ch << '\n';
```

However, the switch version is arguably neater in this case. In general, preference should be given to the switch version when possible. The if-else approach should be reserved for situation where a switch cannot do the job