CHAPTER 2

Control Structures

Prior to program development for a particular problem, the developer should have a clear understanding of the problem definition and a planned approach towards solving the problem on hand. In the following section, we shall describe algorithms and pseudocodes that are prime requirements for program development.

2.1 Algorithms and Pseudocodes

Computing problems can be solved by executing a series of actions in a specific order. A procedure for solving a problem in terms of the actions to be executed and the order in which these actions are to be executed is referred to as an algorithm. To formally define algorithm, it is a sequence of well-defined computational steps that needs to be executed in some order to transform the user input to the desired output. It can be viewed as a transformation function. Specifying the order of executions of the various statements that make up a program is called as program control. A pseudocode is an artificial and an informal language that helps programmers develop algorithms.

Pseudocode is similar to everyday English. It is convenient and user-friendly and is not an actual programming language. Pseudocode is not executed on computers. They help the programmers to logically think out a program before writing or developing the program confirming to the language. Pseudocode should include only executable statements [with the program in mind]. Variable declarations that instruct the compiler to reserve memory space, those that does not cause any input or output action during program execution should not be included in pseudocode.

2.2 Control Structures

Generally, program statements are executed sequentially [one after another], also referred to as sequential program execution. Execution need not be always in sequence, the next statement to be executed can be a statement that is not the next in sequence. This is referred to as transfer of control. Programs are normally developed with three control structures, namely the sequence, selection and repetition structures. The default mode of execution is sequential. However, this sequential flow can be altered with the remaining two control structures available.

Algorithms [text-description] and flow charts [pictorial representation] are normally used in an interchanging fashion before program development. Both of them address the issue of chalking out a solution strategy for the problem on hand in an an easy-to-understand format.

not restricting oneself to the programming language features. It is one among the two that is chosen as the first step of program development. For the convenience of beginners we shall towards the end of this section describe the algorithm, flow chart and C/C++ code for a particular problem. There are three types of selection structures available in C/C++ as follows:

- The `if` selection structure either performs [selects] an action if a condition is true, or skips the action if the condition is false. This structure is also referred to as single selection structure.
- The `if-else` selection structure performs an action if a condition is true [if part], and performs a different condition action if the condition is false [else part]. This structure is also referred to as double selection structure.
- The `switch` selection structure performs one of many different actions possible based on the value of an integer or integer expression. This structure is also referred to as multiple selection structure.

C/C++ provides three types of repetition structures, namely `for`, `while` and `do-while`. Each of these words is called a keyword [interested readers refer to Appendix for a list of C/C++ keywords]. Keywords should be made use [typed-in] the same manner as they appear in the keyword list, failing which would cause syntax errors. Thus, there are in all seven control structures, namely `sequence`, `if`, `if-else`, `switch`, `for`, `while`, `do-while`. These control structures can be manipulated [made use of in different or varying combinations of one another] to implement the algorithm.

Programs developed using control structures usually are single entry and single exit based. Control structures may be connected with each other [the exit point of one control structure is connected to the entry point of the next control structure]. This is nothing but control structure stacking [referring to the accumulation of control structures one after the other, top be literal, it is stacking of control structures one above the other]. The only other form of manipulation control structures is nesting them or placing one control structure inside or within another structure. This is referred to as control structure nesting. This chapter will be dedicated to discuss these control structures and possible ways of manipulation in an exhaustive manner with easy-to-understand examples.

2.2.1 Selection Structures in C/C++

The `if` Selection Structure

A selection structure is required to choose one among many alternative courses of action. For example, suppose that the passing marks in an exam is 45, then the pseudocode statement would be: *if student's marks is greater than or equal to 45*, then *display the student has passed in the examination.*

The above pseudocode statement determines if the condition [mark greater than or equal to 45] is TRUE or FALSE. If the condition is true, then the string, the student has passed in the examination, is displayed on the screen and then the next pseudocode statement is performed. If the condition is FALSE, then the display statement is ignored and the next pseudocode statement in order is performed. Readers kindly note the space left before the

display statement. It is referred to as indentations and should be done with source codes. It is a good coding practice to indent control structures. They promote program readability and clarity. The compiler ignores such blank spaces. Even though it is not a syntactic mandation to indent control structure, the author shall take the readers with the confidence to stick on to the diplomatic syntax of indenting control structures. The above pseudocode statements when translated in C looks like

```
if (stud_mark >= 45)
printf("The student has passed in the examination \n");
```

As we have already explained, the condition within the `if()` is checked. If the condition evaluates to TRUE (non-zero value), then the action specified within the `if` is performed [the corresponding `printf` statement gets executed]. Note the indentation that has been provided for the `printf` statement, this clearly identifies that it belongs to the `if` part. Indentations do not cause any performance enhancement. It is just to improve program clarity and for better understanding. The equivalent flow chart representation for the `if` selection structure is as shown in Figure 2.1.

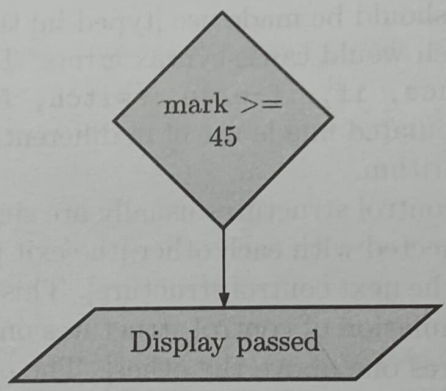

Figure 2.1: Flow chart for the `if` selection structure.

The flow chart consists of an important flow-charting symbol [diamond-shaped] called the decision symbol, used to pictorially represent the decision-making process. To review the other most commonly used flow chart symbols. Oval symbol with start or end labels within it represent the beginning or start and termination or end of an algorithm. [.] Circle symbol used as page connectors, used when a flow chart exceeds within a page limit. [A—made use of in the previous page as last symbol and in the subsequent page as first symbol.]

A parallelogram symbol is used for representing input and output statements [Read Mark]—flow indicator or sequence indicator. Down arrows represent the control flow from the entry till the exit point. Rectangle symbol to represent processing statements or actions `[sum = a + b]`.

We have already stated that the condition of a control structure is more of an expression constructed using relational or equality operators. The decision is made based on the value of the expression; an expression that evaluates to a non-zero value is treated as true value. A zero value is treated as false.

C++ provides the data type Bool to represent values of true or false. This is not available with C.

if-else Selection Structure

The if selection structure discussed earlier performs the required action only if the condition is true, failing which it is skipped. There is no way of specifying an alternate course of action if the condition is false. It is provided in the `if-else` construct. The same pseudocode example taken earlier when extended and coded with the `if-else` construct looks in C like

```
if (std_mark >= 45)
printf("Passed \n");
else
printf("Failed \n");
```

An additional conditional operator available in C/C++ is `?:` [ternary operator]. The operator takes three operands. The operands together with the conditional operator form a conditional expression. The first operand is a condition, the second operand is the value for the entire conditional expression if the condition evaluates to true, and the third operand is the value when the condition evaluates to false. For example, the statement `result = (mark >= 45 ?Passed:Failed);` assuming that the result variable is declared as a string, the condition (expression 1) `mark >= 45` is evaluated. If it evaluates to true, then expression `2[Passed]` is evaluated and the result of this evaluation is assigned to the LHS variable result. Else expression `3[Failed]` is evaluated and its value is assigned to the result. In this case, expressions 2 and 3 are more of values rather than conditional expressions. Thus, depending on the mark either or the string `passed` or `failed` is stored in the result variable. The value in a conditional expression can also be statements or actions to be executed. For example, the above statement can be remodelled as follows:

```
mark > = 45 ?  printf("Passed"):printf("Failed");
```

As explained above, the condition `mark >=45` is evaluated. If the condition evaluates to true, then the condition, in this case, the action of redirecting string `Passed` to the screen is executed. If the condition evaluates to false, then the string `Failed` is redirected to the screen.

Nested `if-else` structures test for multiple cases by placing a `if-else` selection structure inside another `if-else` selection structure. The following piece of code carries out 3 conditional checks on mark and displays the corresponding statements:

```
if (mark > = 90)
printf("Excellent \n");
else
if (mark >= 70)
printf("Average \n");
else
if (mark >= 45)
printf("Passed \n");
else
printf("Failed \n");
```

The above style of indenting may cause the programmer to run out of space on a line and single line source codes to be split across multiple lines on the editor window. To avoid this, an equivalent representation shown below can be used.

```
if(mark >= 90)
printf("Excellent \n"); ⇒ 1
else if (mark >= 70)
printf("Average \n") ⇒ 2
else if (mark >= 45)
printf("Passed \n"); ⇒ 3
else
printf("Failed \n"); ⇒ 4
```

If mark is greater than or equal to 90, then 1 is executed and the remaining statements are skipped. If `mark >= 90` evaluates to false, then the second condition `mark >= 70` is checked. If this evaluates to true, then 2 is executed. The pattern follows similarly. The last `printf(4)` is executed when all the preceding conditions evaluate to false.

Note: A nested `if-else` structure can be much faster than a series of single selection, if structures because of the possibility of early exit after any one of the conditions are satisfied. Hence, it is a good programming practice to test for conditions that are likely to be true at the beginning of the nested `if-else` structure enabling faster execution and early exit than will testing infrequently occurring cases first.

The `if` selection structure expects only one statement in its body. To include several statements in the `if` body or `else` body, the statements should be enclosed between open and close curly braces [{ }]. A set of statements enclosed with a pair of braces is called as compound statement. This can be interpreted as forming a block. An example for `if-else` selection structure with compound statements is as follows:

```
if (percentage >= 45)
{
printf("Passed \n");
printf("Can be promoted to higher grade \n");
}
else
{
printf("Failed \n");
printf("Should repeat the course \n");
}
```

Depending on the value of the condition, the two statements that form the `if` or the `else` block are executed. Statements that are not enclosed within braces but succeed the `if` or `else` block will get executed always irrespective of the condition value [based on the sequential flow of program control].

Common programming errors that are possible with selection structures when coded in C/C++ are as follows:

Missing out on one or both the braces of a compound statement may cause syntax or logical errors in the program. Placing a semicolon after the condition in a single selection structure causes logical error and syntax error in double selection structures.

To avoid such errors it is a good programming practice to type the braces of a compound statement first even before the individual statements comprising the block are keyed in. This

prevents accidental omissions of braces. A compound statement can contain declaration as well, in which case only the compound statement is referred to actually as a block. The syntactic compulsion in C is that all variable declarations should appear at the beginning of the block, before any executable statement of the block [`main()` or any other block]. However, C++ does not impose this syntactic restriction, it allows what is called on the place variable declarations or declaration of variables whenever they are needed is allowed. That is to say variable declarations and executable statements can alternate in C++.

2.3 The while Repetition Structure

A repetition structure allows the programmer to specify that an action is to be repeated on some condition being true. Thus, it is conditional repetition that is performed more often with repetitive control structures available in C/C++. The syntax for the `while` structure is as follows:

```
while (condition)
{
.
.statements
.
}
```

The list of statements mentioned within the `while` block should at some stage or the other make the condition of the `while` block to evaluate to false failing which it causes infinite looping or looping for ever.

Example:

```
counter = 2;
while (counter <= 10)
{
printf("Example \n");
printf("while - loops \n");
counter =counter + 2;
}
```

In this case, the repetition is carried out five times. The condition to be satisfied for repetitive execution is `counter <= 10`. This being true the statement or block of statements that forms the `while` loop construct is executed repetitively. As we have stated earlier, the statement `counter = counter + 2` takes care of modifying the counter values [each time incremented by two]. At some stage or the other counter value would exceed 10, and this serves as the terminating condition for the repetitive structure. The count of 5 repetitions is arrived at as shown in Table 2.1.

Program execution resumes with the first statement after the close brace of the `while` block [sequential execution]. The flow line emerging from the rectangular block wraps back to the decision [condition check] that is tested each time through the loop until the condition becomes false.

Table 2.1: Example for counters

Counter value	*Count of repetitions*
2	1
4	2
6	3
8	4
10	5
12	Loop terminated

2.3.1 An Example Program

Problem: A class of 20 students took an exam on C programming. Compute the class average for the above course.
Solution: The class average is the sum of the marks divided by the number of students. The algorithm for solving this problem must accept the marks [input], perform averaging [processing] and then finally display the result [output].

The pseudocode for the above problem is as follows:

1. Set total to zero
2. Set markcounter to 1
3. `while` markcounter is `<= 20`
4. Input next mark
5. Add this mark to total
6. Add one to markcounter
7. Set class average to total divided by 20
8. Print the class average

This form of repetition is called counter-controlled repetition or definite repetition since the number of repetitions is known well in advance. The C source code for the above algorithm is shown in Figure 2.2.

Note that all the variable declarations precede the first executable statement of the main block. The average variable can also be declared as float to reflect floating-point averages. Variables for whom values will not be entered by the user but those that will be made use of in mathematical calculations or those that will be referenced later should be initialized to valid starting point values. In this case, the variables markcounter and total are initialized to 1 and 0. If such variables are not initialized they may contain values that were assigned their memory locations as a part of a previous program in execution. Such variables called automatic (auto) variables, get initialized with JUNK or GARBAGE [junk values if not user initialized]. Storage classes will be dealt with in detail in Chapter 3.

Note: In a counter-controlled repetition, because loop counter is incremented, using the counter value after looping, is referred to as an off by one, or off by increment error, assuming the increment is in steps of one.

```
1. #include<stdio.h>
2. int main()
3. {
4. int total=0,markcounter=1,mark,average;
5. while ( markcounter<= 20)
6. {
7. printf("\n Enter Mark");
8. scanf("%d",&mark);
9. total=total + mark;
10. markcounter=markcounter + 1;
11. }
12. average=total/20;
13. printf("class average is %d",average);
14. return 0;
15. }
```

Note: In the above code:

Variable	Represents
total	sum of marks
markcounter	number of marks
mark	mark obtained
average	class average

Figure 2.2: C code to compute average.

2.4 Sentinel-controlled Repetition

Counter-controlled repetition can be the choice only if the number of repetitions is known in advance. What if in case the above program is to be used to compute the average of a class consisting of `n` students, `n` being an arbitrary number. A sentinel also called a signal or dummy flag value which indicates end of data entry. In this case, mark can be used as a sentinel value and the sentinel check on it could be `(mark ! =-1)`. As long as value for mark variable entered by the user is not –1 (end of data entry) marks are accepted and the cumulative total is computed. One extension would be to count the number of marks entered by the user [would be required to compute the average] as the class strength is not known in advance and it is arbitrary.

The modified code looks like

```
printf("Enter value for mark \n");
scanf("%d",&mark);
while (mark !=-1)
{
total=total + mark;
markcounter =markcounter+1;
printf("Enter the next mark \n");
scanf ("%d",&mark);
}
```

Note: Programs that make use or sentinel-controlled repetition should at each stage of data entry clearly remind the end user about the sentinel value. And also there should be prompt statements to clearly indicate to the end user the input type and guide him/her in the input process.

The above code to reflect floating average would declare the variable average of float data type. However, still the result of `average = total / n;` will not reflect the fractional part since the result of an integer division [2 integers being divided] is always an integer and the fractional part is truncated. It is this division that is performed first and then assigned to the average variable. Thus, redeclaring average alone does not solve the purpose. To come out of this C, C++ supports type casting, an operation that creates temporary variables of the data type required.

The following statement `float average = (float) total/n` with average declared as float converts the integer variable total temporarily to type float. The result of a float divided by an integer quantity is a float and the value is assigned to the float variable average. Type casting or type conversion is only a temporary (for the operation) conversion and further references (excluding the type casting) the variable is accessed with the initial declaration in place or in effect.

Note: Choosing a valid data value as sentinel-value can cause serious logic errors. Most programs can be viewed as collection of three phases, namely initialization, processing and termination phases. Initializing variables at the point of declaration avoids the problem of uninitialized data. A good programming practice is to go through an exhaustive pseudocode or algorithmic phase after which development of code is straightforward.

2.5 Assignment Operators

These operators are made use of in assignment statements to assign values to variables. In addition to the naive (=) assignment operator, let us look at the other assignment operators available in C/C++. The += is an abbreviated assignment operator. For example, the statement `a + = 3` expands as `a = a + 3` and performs the assignment of variable a incremented by three to the variable a itself. In fact, any statement of the form variable = variable operator expression can be written as variable operator = expression.

The advantage of using abbreviated operators is that certain compilers generate codes that run faster when abbreviated assignment operators are used.

2.5.1 Increment–Decrement Operators

C/C++ also provides the unary increment (++) and decrement (--) operators. These operators can be used to increment or decrement variables by 1. An increment/decrement operator placed before a variable is called preincrement/predecrement operator. An increment /decrement operator placed after a variable is called postincrement/postdecrement operator. The difference between the pre- and post-versions lies in when the assignment and increment/decrement operations are performed.

Unary increment (++) and decrement (--) operators are an abbreviated form for increment/decrement by 1 and then assignment. For example, `a++` expands as `a=a + 1` and `a--` expands as `a = a - 1`. Now, `++a` and `--a` also expand as `a = a + 1` and `a = a - 1`.

Then where lies the difference. The postincrement/postdecrement operator first executes the assignment and then performs the increment/decrement operations. The preversion first increments/decrements and then assigns the incremented/decremented variable to the LHS variable. In other words, Preincrement/Predecrement→ First, Increment/Decrement, and then Assign Postincrement/Postdecrement→First Assign, and then Increment/Decrement.

For better understanding let us consider the following numeric example shown in Figure 2.3.

```
1.  #include<stdio.h>
2.  int main()
3.  {
4.  int a=5,b=6,c=0,d=0,e=0,f=0;
5.  c = a++;
6.  d = ++a;
7.  e = b--;
8.  f = --b;
9.  printf("Values after manipulations are
    %d %d %d %d %d %d",a,b,c,d,e,f);
10. return 0;
11. }
```

Figure 2.3: C code to illustrate increment and decrement operators.

- **Line 5:** `c = a++;` postincrement value of `a`, that is assigned the current value of `a` to `c` and then increment `a` by 1. Therefore, line 5 performs `c = 5` and `a = 6`.
- **Line 6:** `d = ++ a;` preincrement value of `a`, that is incremented the current value of `a` by 1 and then assigned this incremented value to `d`. Therefore, line 6 performs `a = 7` [6+1] and `d = 7`.
- **Line 7:** `e = b--;` performs `e = 6` and `b = 5`.
- **Line 8:** `f = --b;` performs `b = 4` and `f = 4`. Thus, the final output looks like: values after manipulations are 7 4 5 7 6 4.

Note: Unary operators should be placed next to their variables without any intervening spaces. It is just to improve code readability. Incrementing/decrementing a variable in a statement by itself, the pre and postversions have the same effect. It is only when a variable appears in the context of an expression involving other variables that pre and postincrement/decrement operators have different effects. In other words, `a++` and `++a` when used as a statement have the net effect of incrementing `a` by 1, with no differentiation between pre- and postincrement.

Attempting to apply increment or decrement on an expression rather than on a variable such as `++(a+5)` will cause syntax errors. The `++, --, + , -` (unary operators)after parentheses in the operator precedence table. Postincrement and postdecrement are before their precounterparts and associate left–right, whereas the preoperators associate right–left.

2.6 The for Repetition Structure

All features of counter-controlled repetition are provided with the `for` construct. The general structure of a `for` loop is for (initalizer; terminator; increment/decrement) The initializer takes care of initializing the loop counter variable to a valid value. The terminator checks the condition for further looping and to decide on the terminating or stopping criterion. The increment/decrement takes care of modifying the loop counter variable by the specified number of steps. The `while` loop of the earlier counter-controlled repetition example when replaced with a `for` loop looks as follows:

```
for (markcounter=1; markcounter <= 10;markcounter++)
```

The loop counter initialization and loop counter modification (`markcounter=markcounter+1`) can be excluded from the code when using a `for` loop. The `for` structure specifies each of the items needed for counter-controlled repetition. If the body of the `for` loop has more than one statement, then the statements should be enclosed within braces [{ }]. C has the requirement that loop counter/control variables also be part of the initial declaration statements. But C++ allows on the place declarations which means as a part of the initializer of a `for` header structure counter variables may get declared. But such variables remain accessible only from within the `for` loop. This restricted use of control variable name is called variables scope.

Scope refers to referential capability of a variable in a program which will be explained in Chapter 3 to follow. Sometimes the initialization and increment/decrement expressions can be comma-separated. Comma operator has the lowest precedence. The value and type of a comma-separated list of expressions is the value and type of the rightmost expression in the list. Comma operators are often used in `for` loops. It aids in providing multiple initializers or increment and decrement expressions when more than once control variable controls the `for` loop. A list of expressions delimited by comma operator associates left to right and within an expression the result/value is the rightmost value.

The three expressions in the `for` structure are optional. If the terminator or the loop continuation test is omitted, C/C++ assumes that the loop continuation condition is true and creates an infinite loop. Initializer can be omitted assuming that the loop counter or control variable is initialized somewhere else in the program. The increment or decrement operation can as well be performed within the body of the `for` or if no increment is needed and thus gets excluded from the `for` header structure. Common programming errors that can occur with for loops are: commas used instead of semicolons in the `for` header cause syntax error.

A semicolon placed after the right parentheses of the `for` header causes the body of the `for` structure to be treated as an empty statement, and causes logical error. This is sometimes used to create a delay loop that does nothing but the counting operation. This delay can be used to provide a time gap when the output operations are otherwise too quick for the end user visibility.

The initializer, terminator and increment of a `for` header can also be expressions. It is valid to have a `for` structure such as: `for (j = x; j <= 4 * x * y; j += y / x)`, assuming `x=2, y=10`, the loop structure reduces as for `(j=2; i <= 80 ; j = j+5)`.

In the above `for` loop, the increment is done in steps of 5 each time. In general, the increment operation of a `for` structure can also be replaced with a decrement operation in which case the loop counts in the downward direction. If the loop continuation condition is

false to start with, then execution proceeds from the statements following the `for`.

A few other examples:

1. Vary the control variable from 5 to 55 in steps of 5 for `(i = 5; i <= 55; i += 5)`.
2. Vary the control variable from 100 to 1 in step of 1 for `(i = 100; i >= 1; i --)`.

A simple application of `for` loop to compute the sum of odd integers from 3 to 99 is shown in Figure 2.4.

```
1. #include<stdio.h>
2. int main()
3. {
4. int i=3, sum = 0;
5. for ( i = 3; i <= 99; i + = 2)
6. sum + = i;
7. printf("Sum of odd integers between 3 & 99 is %d",sum);
8. return 0;
9. }
```

Output:
Sum of odd integers between 3 & 99 is 2499

Figure 2.4: C code to compute sum of odd integers.

2.7 Good Programming Practices with `for` Loop

Avoid merging the `for` body with the `for` header even though it is possible syntactically. This reduces program clarity and makes programs difficult to understand. Limit the size of control structures to a single line of code (LoC). Avoid placing expressions whose values do not change inside loops, but then today's sophisticated compilers can remove such statements and place them outside the loop in the machine language code. This phase is referred to as Loop Optimization.

2.8 The `switch` (Multiple) Selection Structure

The `switch` selection structure is used when more than two alternative actions are possible, and one from the list of actions needs to be performed based on some integral values. The `switch` structure consists of a series of case labels and an optional default case. An example program to understand the syntax and semantics of the `switch`-case construct is shown in Figure 2.5. Develop a program to perform different arithmetic operations of `+,-,/,*` on two numbers input by the user based on a user choice.

Depending on the choice entered by the user the case labels [which can be integer or character] are compared and the case is chosen for execution. Cases actually correspond to the multiple courses of action possible with a programming logic. A program that has five different actions based on the value of a control variable will have five case labels in its

```
1. #include<stdio.h>
2. int main()
3. {
4. int number1, number2, choice, result;
5. printf("Enter number 1");
6. scanf("%d", &number1);
7. printf("\n Enter number 2");
8. scanf("%d", &number2);
9. printf("\n Enter your choice \n");
10. printf("1.Addition \n 2.Subtraction \n 3.Multiplication
    \n 4.Division");
11. scanf ("%d", &choice);
12. switch(choice)
13. {
14. case 1: result = number 1 + number 2;
15.    printf("The sum of %d & %d is %d",number1,number2,result);
16.    break;
17. case 2: result = number 1 - number 2;
18.    printf("The diff. of %d & %d is %d",number1,number2,result);
19.    break;
20. case 3: result = number 1 * number 2;
21.    printf("The prod. of%d&%d is %d",number1,number2,result);
22.    break;
23. case 4: result = number 1 / number 2;
24.    printf("The quot.of %d / %d is %d",number1,number2,result);
25.    break;
26. default: printf("Invalid Choice , Exiting \n");
27.    break;
28. }/*End of switch*/
29. }/*End of main*/
```

Figure 2.5: C code to illustrate use of `switch` construct.

`switch-case` construct. The default case is chosen for execution when the choice entered by the user matches none of the expected values. A block of statements to be treated as a single case need not be enclosed within braces ( ).

We resume back to our definition of `switch` selection structure; it is choosing one action from a list of multiple actions possible. So at any point of time it is only one action that can be performed. After having selected or performed one action there should be some way of quitting the remaining list of actions. The `break` statement precisely performs this. The `break` statement causes program control to proceed with the first statement after the `switch` structure. Missing out on `break` statement(s) will cause program control to proceed to the following cases until the compiler encounters a break in anyone of the following cases. The break statement literally breaks program control from the `switch` structure. This feature is exploited to perform the same action for several cases.

Possible programming errors with switch structures are: omitting the break statement can cause logical errors.Omitting the space between the keyword case and its label can cause a logic error by creating an unused case label. For example, instead of case 10, when coded as case 10, it will not perform the appropriate actions when the control variable has the value of 10. Providing identical case labels within the switch structure can cause syntax error.

A break statement may not be required with the default case if it is listed as the last case. However there is no syntactic compulsion regarding the ordering of the cases within a switch structure. Note that if the switch control variable is of type char, at run time to have the program read characters they are sent to the computer by the pressing of enter key on the keyboard. This places a new line character after the character is processed. This has to be handled separately to make the program work correctly by including a skipping logic with these special characters (\n,\t) [that is empty case statement]. Doing so one can prevent the default case getting intermixed with such special cases.

2.9 The do-while Repetition Structure

The do-while is similar to the while control structure. In while structure, the loop continuation condition is tested at the beginning of the loop or before the body of the loop is executed, the do-while constructs checks for the loop continuation condition after executing the loop body. Thus, the loop is executed at least once. A do-while on termination, execution continues with the statements after the while clause. Other aspects of the do-while and the while repetition structures are the same. do-while can be the choice in cases a sequence of actions will always be performed once and then based on some condition (sentinel value) the same sequence either gets repeated or execution proceeds with the next set of actions.

Syntax for do-while is as follows:

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

A do-while statement with no braces around (syntactically allowed) in a single statement body of the loop appears as follows:

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

This can cause the reader to misinterpret the statement while (condition), as an empty while construct which clearly is not the case. Hence to avoid this, it is a good practice to always have braces for do-while construct of the form mentioned earlier. Thus, all the set of actions of a do-while construct gets executed at least once.

2.10 break and continue Statements

These statements alter the flow of program control. The `break` statement when made use of within repetitive or control structures such as `for/do-while/while/switch` causes immediate exit from that structure. Program execution continues with the first statement after that structure. The `continue` statement used with repetitive control structures skips the remaining statement(s) after the `continue` statement and proceeds with the next iteration of the loop. In `while` and `do-while` structures the loop continuation condition is evaluated immediately after `continue` statement, while with `for` structure the increment is first executed and then the loop continuation condition is executed. The next exercise will explore on the application of `break` and `continue` statements within a program and its effect on program execution. `break` and `continue` statements do alter the flow of program control to some extent, as will be clear from the following example shown in Figures 2.6 and 2.7.

```
 1.  #include<stdio.h>
 2.  int main()
 3.  {
 4.  int x;
 5.  for(x = 1 ; x <= 20; x ++)
 6.  {
 7.  if(x == 5)
 8.  break;
 9.  printf("%d",x);
10.  }
11.  printf("Exited loop @ %d",x);
12.  return 0;
13.  }
```

Output of the above code:

1 2 3 4 Exited Loop at 5.

Figure 2.6: C code to explain the use of `break`.

2.11 Logical Operators

The C/C++ logical operator can be exploited to form complex conditions that can be used in control structures. The logical operators of `&&(AND)`, `||(OR)`, `!(NOT)` are used to frame complex conditions. The logical `AND (&&)` operator can be used to test if all the conditions (more than 1) are TRUE or not [if `(a==5 && b==8)` C/C++ code$\cdots$].

Logical `AND` first evaluates the leftmost condition. Only if this evaluates to true, does it proceed evaluating the remaining conditions? This is based on the principle that all conditions should evaluate to true for the output of logical `AND` to be true. This optimized logical `AND` is also commonly referred to as short-circuiting. For a better understanding the truth table of a two input `AND` operation is shown in Table 2.2.

```
1.  #include<stdio.h>
2.  int main()
3.  {
4.  int x;
5.  for(x = 1 ; x <= 10; x ++)
6.  {
7.  if(x == 5)
8.  continue;
9.  printf("%d",x);
10. }
11. printf("Control here") ;
12. return 0;
13. }
```

Output:

1 2 3 4 6 7 8 9 10 Control here.

Figure 2.7: C code to explain the use of `continue`.

Table 2.2: Logical `AND` truth table

Expression 1/Condition 1	*Expression 2/Condition 2*	*Result*
TRUE	TRUE	TRUE
FALSE	TRUE	FALSE
TRUE	FALSE	FALSE
FALSE	FALSE	FALSE

Logical `OR` (||—double pipe symbol) is used to check if at least one among the several conditions is TRUE or not. The truth table is shown in Table 2.3.

Table 2.3: Logical `OR` truth table

Expression 1/Condition 1	*Expression 2/Condition 2*	*Result*
TRUE	TRUE	TRUE
FALSE	TRUE	TRUE
TRUE	FALSE	TRUE
FALSE	FALSE	FALSE

Both logical `AND` and `OR` are binary operators. Thus it is only any two conditions that are checked for trueness at any time. Multiple condition checks using `&&`,| textbar proceed left-right combining the first-two conditions and then the result of this operation is logically `AND`ed or `OR`ed with further conditions. The remaining logical operator (`!` - `NOT`) is used to test the reverse meaning of a condition. It is a unary operator and is called logical negation operator. It is placed before a condition when the programmer wants to choose a path on the original condition being false. For example:

```
if ( !  ( x == y ) )
printf("Example \n");
```

The parentheses are important because the ! operator has higher precedence than the quality operator. Otherwise, the expression will be parsed by the compiler as (! X == Y) which is not what we want. Common logic errors that can occur with (=) and (==) operators are as follows:

Often programmers accidentally swap the equality (==) and the assignment (=) operator. Unfortunately, this does not cause syntax error, but can lead to serious logical errors. An example to understand this concept:

```
if (a == 0)
printf("Success \n");
else
printf("Failure \n");
```

The output on the screen is failure. How? Line 2, an intended comparison as a result of the missing = is performed as an assignment operation. It proceeds as an assignment operation. It proceeds as if (0). The result of the condition, 0 [false in C/C++] causes the else part to be executed even though the value of a is equal to 0. Any non-zero value returned by a condition is treated as true and zero value is treated as false. And more worse the equality operator (used for comparison) is more of a read operation. When this is accidentally swapped with the assignment operator (=) it amounts to the LHS variable having its value changed, an operation that would not have been intended by the programmer.

One diplomatic solution for overcoming this problem would be to code comparison/equality statements as constant value == variable name, e.g. 10 == a in an if condition would solve our problem because an accidental miss of an = in the above format would be treated as 10 = a; assignment operators expected value to be a variable, which is clearly violated here and hence the error gets caught at the compilation stage itself. Well, this solution is valid only as long as variable is compared with a constant value. But once when two variables need to be compared or checked for equality or compared the above solution solves no purpose. But then more often it is with constant values that variables are checked for equality. The case when two variables are compared, well, there is no way out for tracing this logical error. Programmer will have to live with it but then this arises only due to programmer's slack coding practices.

Now, the other scenario when == is used in the place of =, such as X == 5; again this does not lead to syntax errors. It proceeds as a comparison operation and the value returned is true. Irrespective of true or false is returned the value 5 is lost once for all! Unfortunately, there is no way out for this problem as we had for the earlier one.

Review Questions

1. Differentiate an algorithm from a source code.
2. Interpret the term control structure and name the three control structures available in C++.
3. Name a few object-oriented languages other than C++ available in the market today.
4. Develop a C++ program to compute the maximum and minimum of three integers accepted from the user.

5. Develop a C++ program that accepts marks scored by a student in six subjects and display whether he/she has passed or not. Assume a passing minimum. Also display the average in case the student has passed in all subjects.
6. Develop a C++ program that checks if a number input by the user is an (i) armstrong, (ii) adams number or not. An armstrong number is one in which the sum of cubes of the individual digits of the number equates to the original number. An adams number is one in which the reverse of the square of the number is the same as that of square of the reverse of the number. Example for armstrong number is 407, where $4^3 + 0^3 + 7^3 = 407$, and adams number is 12, where square of 12 is 144 and reverse is 441, which is the same as square of 21 (reverse of 12).
7. Develop a C++ program that generates the list of all armstrong and adams numbers between a user-specified range. Accept user-specified lower and upper limit.
8. Develop a C++ program to check if an user input integer is a palindrome or not. A palindrome is an integer that reads the same forward and reverse. Example is 1441.
9. Differentiate the break and continue statements with suitable examples (other than the one used in the text).
10. Develop a C++ program that generates the multiples of 2 within a user-specified range. Display the square, cube and square of numbers within a user-specified range in a tabular column format.
11. Develop a C++ program to compute the factorial of a number using both increment and decrement loops.
12. Develop a C++ program to perform decimal to binary, octal and hexadecimal format conversion and vice versa using `switch` case construct.
13. Develop a C++ program that computes the values of e^x.
14. Explain the effect of a misused assignment operator as a part of a conditional statement within an `if` statement.
15. Visit the directory in Unix/Linux where header files used in C and C++ are stored and appreciate the contents of them.