LESSON 3
CONTROL STRUCTURES

PROF. JOHN P. BAUGH
PROFJPBAUGH@GMAIL.COM
PROFJPBAUGH.COM

CONTENTS

INTRODUCTION

In this lesson, we'll explore the topic of **control structures**. Primarily, we will focus on **repetition control structures**, also known as **iterative control structures** as well as **branching control structures**, also known as **selection control structures**. This chapter will help us establish a solid foundation to write significantly more complex programs, by allowing the program to make simple decisions through its execution depending on the state of the program, user input, variables, etc.

ASSUMPTIONS

To successfully understand the topics and code in this chapter, you should be comfortable with the topics from lesson 1 and lesson 2. Specifically, you should know how to use variables, arithmetic operators, and retrieve user input. You should also be relatively familiar with how to create a new project in Visual Studio 2010.

3.1 – LOGIC, LOGICAL OPERATORS, AND RELATIONAL OPERATORS

Before we discuss selection and repetition control structures, we must do a brief review of **logic**, as well as the **logical operators**. Additionally, we will look at **relational operators**.

Logic deals with the truth of statements. Using **Boolean algebra**, we can compare and combine statements, even complex ones, to determine the truth or falsity of expressions.

First, we look at the relational operators and examples of how they are applied:

Relational Operator	Name	Description
>	Greater Than	Binary operator that compares its operands in the form a > b. Returns true if a is greater than b, false otherwise.
>=	Greater Than or Equal To	Same as >, but also returns true if the operands are equal
<	Less Than	Binary operator that compares its operands in the form a < b. Returns true if a is less than b, false otherwise.
<=	Less Than or Equal To	Same as <, but also returns true if the operands are equal
==	Is Equal To	Binary operator that compares its operands in the form a == b. Returns true only if a and b are equal. Note that there are *two* equal signs.

!=	Not Equal To	Binary operator that returns true if the operands are not equal. Takes the form a != b.

Also, we may want to combine statements, or even reverse the truth value of certain statements. For this, we utilize the logical operators.

Logical Operator	Name	Description
&&	Logical AND	Takes the form A && B, where A and B may be any logical assertion (simple, such as a > 6, or complex such as another statement using a logical operator.) Returns true if *both* A and B are true, false otherwise.
||	Logical OR	Takes the form A || B, where A and B may be any logical assertion (simple or complex). Returns true if at least one of the two operands is true. Thus, A || B is true if only A is true, or if only B is true, or if both are true.
!	Logical NOT	This is a unary operator. In other words, it only takes one operand. The form is !A. Logical NOT negates (reverses) the truth value of a statement.

To fully appreciate the truth values produced by applying logical operators, we use **truth tables**. Given a logical assertion(s) (simple or complex), truth tables let us see the resultant truth value of the compound statement.

3.1.1 - LOGICAL AND (&&) TRUTH TABLE

P	Q	P && Q
true	true	true
true	false	false
false	true	false
false	false	false

As you can see, the P and Q value in the first two columns of the truth table are "what if" values. For example, what is the value of P && Q if P is true and Q is false (the second row) – we see that the answer is "false."

We see that AND returns true on its operands *only* if *both* operands are true. AND returns false otherwise.

3.1.2 - LOGICAL OR (||) TRUTH TABLE

P	Q	P || Q
true	true	true
true	false	true
false	true	true
false	false	false

You should notice that for Logical OR, if *either* (or both) of the operands are true, OR returns true. The only instance in which OR returns false is if both of the operands are false.

3.1.3 - LOGICAL NOT (!) TRUTH TABLE

P	!P
true	false
false	true

Logical NOT is special, in that it is a unary operator (takes only one operand), so its truth table is smaller.

3.2– SELECTION CONTROL STRUCTURES: `if`, `if-else`, AND `switch`

In this section we will discuss the three selection control structures that allow us to make decisions and branch to different code segments depending on some criterion (or criteria.) We will begin with the most simple of the three, the if-statement, and then progress to the if-else-statement and finally, the switch statement.

3.2.1 - THE `if` SINGLE-SELECTION STRUCTURE

An **if-statement**, or **if selection structure** allows us to write a block of code that can be either executed, or not, depending on some special criteria. To use an if-statement, we write the following:

```
if (condition)
{

}
```

You’ll notice that the keyword **if** is used in the header. This is followed by a set of parentheses, with a condition in the middle. The condition is what’s interesting. The condition must evaluation to true or false in order to be valid. Like much involved in programming, it is easiest to learn when given an example:

```
1   #include <iostream>
2   using namespace std;
3   
4   int main()
5   {
6           int weight = 0;
7   
8           cout<<"What is your weight?"<<endl;
9           cin >> weight;
10  
11          cout<<"Welcome to the Gym!"<<endl;
12          if(weight > 300)
13          {
14                  cout<<"You may want to hire a personal trainer."<<endl;
15          }
16  
17          return 0;
18  }
```

Code 3.2-1: Single selection with the if structure

The sample code in ***Code 3.2-1*** illustrates the usage of the if-statement. Let us see the output for two possible inputs:

- 150
- 310

```
What is your weight?
150
Welcome to the Gym!
Press any key to continue . . .
```

Output for Code 3.2-1 with input of 150

```
What is your weight?
310
Welcome to the Gym!
You may want to hire a personal trainer.
Press any key to continue . . .
```

Output for Code 3.2-1 with input of 310

As we can see, regardless of the weight entered, the program prints the statement “Welcome to the Gym!” This is because this code is in sequence and there is no branching applied to it. We have been using **sequential control structure** in our code since the beginning. Statements execute one right after another. The interesting code is present on lines 12 – 15. The header of the if-statement determines if the weight that was input is greater than 300. If it is, then whatever is in the body of the if-statement will execute. This is evident in the two different outputs we tested. Feel free to try values of your own. Specifically, run the program and see what happens when you enter exactly 300. Why does this happen?

3.2.2 - THE `if-else` DOUBLE SELECTION STRUCTURE

As we saw in the last section, the if-statement allows for a block of code to either execute, or not. For this reason, it was called a single selection structure. What if we want one or another thing to happen based on some criterion (or criteria)? In this case, we can use the **if-else double selection structure**.

Again, let's look at some code. You'll find this code to perhaps be practical, and note that something similar is written for various websites and programs.

```
#include <iostream>
using namespace std;

int main()
{
	int age = 0;

	cout<<"What is your age?"<<endl;
	cin>>age;

	if(age >= 21)
	{
		cout<<"Welcome to Snuggleberry's Pub!  Have a drink!"<<endl;
	}
	else
	{
		cout<<"You're under the legal drinking age."
		<<"Sorry, but you can't come in!"<<endl;
	}

	return 0;
}
```

Code 3.2-2: The if-else double selection structure

This gives us the ability to pick between two different options. In ***Code 3.2-2*** specifically, the age entered by the user is either 21 and older, or younger than 21. If the user is 21 or older, the code lets them "into the bar", otherwise it tells them they can't come in.

The output is left as an exercise for the reader.

Another interesting thing that can be done with if-else structure is to create a cascading (or nesting) effect in which if-else statements are chained together. What if we want one of three possibilities of code execution given a specific criterion (or criteria)?

Let us consider the following example:

```
#include <iostream>
using namespace std;

int main()
{
	int age = 0;

	cout<<"What is your age?"<<endl;
	cin>>age;

11          if(age < 16)
12          {
13                  cout<<"You can't even drive yet!"<<endl;
14          }
15          else if (age < 18)
16          {
17                  cout<<"You can drive, but you can't vote yet!"<<endl;
18          }
19          else if (age < 21)
20          {
21                  cout<<"You can drive and vote, but you can't drink yet!"<<endl;
22          }
23          else
24          {
25                  cout<<"You can drive, vote, and drink."<<endl;
26          }
27
28          return 0;
29  }
```

Code 3.2-3

This code, ***Code 3.2-3*** shows what happens if we have more than one option cascaded. Although the if-statements could be nested (one inside another), it is usually clearer to write the code in a cascading fashion. The first block of code under selection control is on lines 11-14. This block of code will execute if the user's age is under 16. The next block of code on lines 15-18 will execute if the user's age is 16 or greater up to 18 (but not including 18.) Then on lines 19-22, if the user is 18 or older (up to 21, but not including 21) the code will execute. Finally, the default code is on lines 23-26, in which case the user is 21 or older.

Notice that the order does matter. With a set of cascading if-statements, the first true if-statement encountered will execute and then the entire chained group of if-statements is done. In other words, even though 15 is less than 16, and also less than 18, and also less than 21, only the first if-statement, whose condition is evaluated on line 11, will be executed.

3.2.3 - THE `switch` MULTIPLE SELECTION STRUCTURE

Although cascading if-else statements does work just fine (in fact, many programmers prefer this style), there are situations in which a structure designed for multiple options might lead to clearer code. The **switch statement** (also known as the **switch-case structure**) is such a structure. Its syntax differs quite greatly from the if-statement and if-else statements that we learned earlier.

```
#include <iostream>
using namespace std;

int main()
{
        char grade;

        cout<<"What is the student's grade?"<<endl;
        cin>>grade;

```

```
11          switch(grade)
12          {
13          case 'A':
14                  cout<<"You're doing excellent in this course!"<<endl;
15                  break;
16          case 'B':
17                  cout<<"You're doing very well in this course!"<<endl;
18                  break;
19          case 'C':
20                  cout<<"You should study more!"<<endl;
21                  break;
22          case 'D':
23                  cout<<"You really need to get a tutor!"<<endl;
24                  break;
25          default:
26                  cout<<"You're failing the course.  :("<<endl;
27  
28          }
29  
30          return 0;
31  }
```

Code 3.2-4: The switch multiple selection structure

The switch statement begins with its header on line 11. This header states what variable should be considered in the case statements within the body of the switch statement.

Here is the output for a couple different runs of the program:

```
What is the student's grade?
A
You're doing excellent in this course!
Press any key to continue . . .
```

Output for Code 3.2-4 with input A

```
What is the student's grade?
F
You're failing the course.  :(
Press any key to continue . . .
```

Output for Code 3.2-4 with input F

These are the results that are expected. But it is possible we may have a slight **logic error** in our code. A logic error is an error that results in incorrect behavior or output given input and/or program state. Let's consider what happens if the user types in a lowercase 'a' instead of an uppercase 'A' for the input.

```
What is the student's grade?
a
You're failing the course.  :(
Press any key to continue . . .
```

Output for Code 3.2-4 with input a

Houston, we have a problem![1]

Why does it say there is a failure, if the student received an A (well, technically, an 'a')? The computer only does what it's told. Our program must make sense of the input explicitly. The computer does not read into the semantics of the code. We want to tell the user they're doing excellent if they have an A (or an 'a' for that matter!) but the problem is 'A' and 'a' are two different characters.

You can cascade case statements by simply leaving out a `break` statement. In fact, this is why the break statement is necessary in the code. It causes the code to exit the switch statement's body entirely. So let's update the code so that lowercase letters are taken into consideration:

```
#include <iostream>
using namespace std;

int main()
{
	char grade;

	cout<<"What is the student's grade?"<<endl;
	cin>>grade;

	switch(grade)
	{
	case 'A':
	case 'a':
		cout<<"You're doing excellent in this course!"<<endl;
		break;
	case 'B':
	case 'b':
		cout<<"You're doing very well in this course!"<<endl;
		break;
	case 'C':
	case 'c':
		cout<<"You should study more!"<<endl;
		break;
	case 'D':
	case 'd':
		cout<<"You really need to get a tutor!"<<endl;
		break;
	default:
		cout<<"You're failing the course.  :("<<endl;

	}

	return 0;
}
```

Code 3.2-5: Correcting a small logic error from Code 3.2-4

Now the code takes into consideration lowercase letters. Now, as an exercise, consider if there is another logic error in the code above, even after our correction. Hint: Can the user really have a 'G' grade or a 'Z' grade? How can you correct this?

[1] "Houston we have a problem"is a statement that was originally attributed to the Apollo 13 astronauts when they experienced technical difficulties in space. Although it is misquoted, it is similar to what was actually said, and has become an American colloquialism to describe any issues/problems that arise in a situation.

3.3 - REPETITION CONTROL STRUCTURES: `while`, `do-while`, AND `for`

So far in this lesson, we've formalized the concept of sequential control, in which statements are executed one after another with no branching. In fact, we've used this control structure since the first lesson. In the last section, we discussed selection control structures, which allow us to branch control, executing or not executing various blocks of code depending on the state of one or more variables.

In this section, we will discuss **repetition control structures**, also known as **iterative control structures**, which allow us to repeatedly execute the same block of code multiple times. Along with sequential and selection structures, repetition will allow us to take further control of our programs.

3.3.1 – THE `while` AND `do-while` REPETITION STRUCTURES

In this section, we will discuss while and do-while repetition control structures. It is probably easier to start with the while control structure, although both are relatively easy to understand.

```
1   #include <iostream>
2   using namespace std;
3
4   int main()
5   {
6           int count = 0;
7
8           while(count < 10)
9           {
10                  cout<<"Count is : "<<count<<endl;
11                  count++;
12          }
13
14          return 0;
15  }
```

Code 3.3-1: ***The while iterative control structure***

```
Count is : 0
Count is : 1
Count is : 2
Count is : 3
Count is : 4
Count is : 5
Count is : 6
Count is : 7
Count is : 8
Count is : 9
Press any key to continue . . .
```

Output for Code 3.3-1

This example of a **while control structure**, more commonly referred to as a **while loop**, contains a header on line 8. This header is similar to what we saw in the previous section when we explored if-statements. In this context, the body of the while loop (lines 9 – 12) will execute as long as the condition in the while loop's header remains true. This condition is sometimes referred to as the **loop-continuation statement**. This holds the condition that must remain satisfied in order for the loop to continue.

This type of loop is referred to as a **count-controlled loop**. This is because it loops a certain number of times, based on a count value of some sort. Notice that we initialize the `count` variable on line 6 outside of the loop. Then, near the end of the while code block, on line 11, we increment `count`. This is important, because the `count` variable must move toward the condition which will terminate the loop. Otherwise, we'd end up with an **infinite loop**, that is, a loop that does not terminate (or at least, doesn't terminate under normal circumstances.)

Another repetition structure, the **do-while control structure** is very similar to the while control structure except that a do-while loop is guaranteed to execute at least once. Let's see this in action.

```
1   #include <iostream>
2   using namespace std;
3
4   int main()
5   {
6           int count = 10;
7
8           while(count < 10)
9           {
10                  cout<<"Count is : "<<count<<endl;
11                  count++;
12          }
13
14          do
15          {
16                  cout<<"do-while count is: "<<count<<endl;
17                  count++;
18          }
19          while(count < 10);
20
21          return 0;
22  }
```

Code 3.3-2: ***while vs do-while loops***

The output is as follows:

```
do-while count is: 10
Press any key to continue . . .
```

Output for Code 3.3-2

Note that the counter is initialized to 10 on line 6 in ***Code 3.3-2***. In this case, on line 8, the condition `count < 10` is not satisfied. Thus, the loop is never entered. Then, on lines 14 – 18, the body of the do-while loop is executed once, and *then* the loop continuation condition is checked on line 19. Since the condition is false, the loop exits.

But note that the do-while loop did get in one execution cycle before termination. Thus, its behavior is slightly different from the while loop.

Another method of using loops involves using a **sentinel value** (also known as a **flag value**.) A typical count controlled loop is also referred to as a **definite loop**, because the number of times it will execute is known. A

sentinel-controlled loop uses a sentinel value and the number of times it will execute is not known ahead of time. Thus, these types of loops are referred to as **indefinite loops**. Note that this is not the same as an infinite loop.

Let us consider an example where a teacher has a program to calculate the average of student grades. The program doesn't know how many students the teacher has ahead of time. So, a special number, -1, represents the sentinel value (in other words, allows the loop to terminate.) In most cases where a sentinel value is used, a **priming read** is required, in which the first input value is entered.

```
1   #include <iostream>
2   using namespace std;
3
4   int main()
5   {
6           int numStudents = 0;
7           double totalScore = 0;
8           double input = 0;
9
10          cout<<"Please enter the first student's score, or -1 to exit."<<endl;
11          cin>>input;
12
13          while(input != -1)
14          {
15                  totalScore += input;   //add the input to the total
16                  numStudents++;       //increment the number of students
17                  cout<<"Please enter the next student's score, or -1 to exit."<<endl;
18                  cin>>input;
19          }//end while
20
21          if(numStudents > 0)
22          {
23                  cout<<"The total of scores is : "<<totalScore<<endl;
24                  cout<<"The average score is : "<<totalScore/numStudents<<endl;
25          }
26
27          return 0;
28  }
```

Code 3.3-3: ***Sentinel control of a loop***

This is one of the largest programs we've written so far, and quite useful, actually. This program has three variables declared on lines 6 – 8. `numStudents` is used to track the number of students. `totalScore`, not surprisingly, is used to total all the scores that the user enters. And finally, `input` is used to receive input before the loop begins, as well as in each iteration of the loop.

After we receive the priming read, the loop begins on line 13. The test for the sentinel value is performed, and then we enter the body of the loop. On line 15, we use the shorthand syntax to add input to the current total score. As you may recall, the += is the **compound addition operator**. Then, we increment the variable `numStudents` on line 16. Then at the end of the loop, on lines 17 and 18, we obtain the data for the next cycle of the loop. If the user enters a -1, then the loop will terminate.

Notice then, outside the loop we have an if-statement. This is necessary, because if the user starts the program, and immediately enters a -1, then the loop will never iterate. Thus, `numStudents` will be 0, and we will perform a division by 0 on line 24. This isn't possible (unless you're Chuck Norris), so we must check to ensure `numStudents` is greater than 0. Note we could have simply typed `!= 0`, but some programmers prefer using > or < in scenarios where other invalid data could have been entered or calculated.

Here is a sample output (note that there could be an infinite number of possible runs of this program, though.)

```
Please enter the first student's score, or -1 to exit.
100
Please enter the next student's score, or -1 to exit.
90
Please enter the next student's score, or -1 to exit.
75
Please enter the next student's score, or -1 to exit.
85.5
Please enter the next student's score, or -1 to exit.
99
Please enter the next student's score, or -1 to exit.
-1
The total of scores is : 449.5
The average score is : 89.9
Press any key to continue . . .
```

Output for Code 3.3-3 with a set of possible input values

3.3.2 – THE `for` REPETITION STRUCTURE

The **for repetition structure**, also known as the **for-loop** is perfectly designed for count-controlled repetition. In the header of the for-loop, the counter variable can be initialized, the condition can be tested, and the counter can be modified.

```cpp
#include <iostream>
using namespace std;

int main()
{

	for(int i = 0; i < 10; i++)
	{
		cout<<"i = "<<i<<endl;
	}

	return 0;
}
```

Code 3.3-4: The for-loop

The code above is self-explanatory, given the previous description. Thus the output is:

```
i = 0
i = 1
i = 2
i = 3
i = 4
i = 5
i = 6
i = 7
i = 8
```

```
i = 9
Press any key to continue . . .
```

Output for Code 3.3-4

3.4 – SUMMARY

In this chapter, we took a look at various control structures. I tried to make you aware of the fact that you' ve been using the most simple control structure, sequential control, for your programs since the very beginning. However, since nearly all programs involve some sort of decision making, I introduced the selection and iterative control structures that C++ has to offer. The selection control structures are if, if-else, and switch. The iterative control structures are while, do-while, and for.

EXERCISES

1. Write a program that calculates the average weight of several input weights. If the user inputs a negative weight, then the program should terminate.

2. Modify the program in Exercise 1 to include a conditional print out – when the user enters a weight 400 lbs or above, the print out should say, "Please encourage this participant to seek nutritional counseling as soon as possible."

3. In many video games, the letters W, A, S, and D are used to control Up, Left, Down, and Right player character movement in the game. Write a program that loops, taking in a character each iteration, and prints out "Moving Up", "Moving Left", etc. depending on the letter that is entered. If the user types the letter 'Q', then the loop should break and the program should exit. Besides W, A, S, D, and Q, any character that is entered should result in the print out, "Invalid command. Please try again."