CIS 278 (CS1) Programming Methods: C++

Lesson 3: Decisions

Skip to Main Content

Section 3.1: The if Statement

Let's go back to our paycheck example from lesson 1, but let's change the rules a bit. (You might want to go back to section 1.5 and review that problem if you have forgotten it.) Suppose we decide to reward our employees who work more than 30 hours a week with a little bonus. From now on these employees will get a $100 bonus in addition to their normal paycheck. To incorporate this into our program we need a way to say "if the employee worked more than 30 hours, add $100 to her paycheck." C++ provides exactly that with the **`if statement`**. Here's how it would work for our example:

```
if (hours > 30) {
    paycheck = paycheck + 100;
}
```

Here's how our program will look with our new addition:

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

int main()
{
    int hours;
    int payRate;
    int paycheck;

    cout << "enter hours worked: ";
    cin >> hours;
    cout << "enter rate of pay: ";
    cin >> payRate;
    paycheck = hours * payRate;

    if (hours > 30) {
        paycheck = paycheck + 100;
    }

    cout << "the amount of the paycheck is "
         << paycheck << " dollars." << endl;
}
```

```
enter hours worked: 31
enter rate of pay: 10
the amount of the paycheck is 410 dollars.
```

The general form of the if statement in C++ is

```
if (<condition>){
    <statements>
}
```

If the condition of the if statement is true, then the statements inside the if statement get executed. If the condition of the if statement is false, then the statements inside the if statement do not get executed. In either case, the program continues execution with the statement following the if statement.

Notice that we are very careful with our indentation here. **The statements inside an if statement are always indented further than the if statement itself**. This is because the statements inside an if statement are **part** of the if statement. If we do not indent them, then it does not look like they are part of the if statement. It looks like they are independent statements that come after the if statement. When you

are about to type a line of code and you are not sure how far it should be indented, ask yourself the question, "Is this statement part of the statement I just typed (for example, a statement inside of an if statement), or does it simply come after the statement I just typed?" If it is part of the statement you just typed, it should be indented further than that statement. If it simply comes after the statement you just typed, it should be indented the same amount as the statement you just typed. You should never indent the next statement after a `cout` statement or a `cin` statement or an assignment statement, because it is impossible for a statement to be "part of" one of these.

I also need to make some comments about the open and close curly braces. First, if there is only one statement inside the if, then you technically do not need the open and close curly brace. Personally I think that it is safer and clearer to just always use the open/close curly brace, even if there is only one statement. But many programmers (perhaps most) like to leave them off when they can. How you write your own code is up to you. However, you certainly need to understand what happens when the curly braces are left off. Several problems on your exams will probably depend on it. For example, consider this code:

```
if (hours > 40)
    statement1;
    statement2;
statement3;
```

In this case, both statement2 and statement3 will get executed regardless of the value of "hours", because statement2 is not inside the if. **Make sure you understand this, and ask me about it if you do not!**

Secondly, I should point out that the text uses a different placement of the open curly brace. In the text that brace is placed down on the next line after the if, instead of at the end of the line. Again, I think the technique used in the lessons is better, but you may choose your favorite and use it. In this case, I'm pretty sure that most professionally written C++ code will follow the style used in the lessons.

If you use a good system, it is always easy to tell whether you have the correct number of close curly braces. In my system (used throughout the lessons), I can always tell by the pattern of the close curly braces whether there are too many or too few. Every line that has an open curly brace at the end MUST have a close curly brace below it somewhere, indented the same amount:

```
if (...) {
|   if (...) {
|   |   if (...) {
|   |   |   statement;
|   |   |   statement;
|   |   }
|   }
}
```

Also, the pattern of the close curly braces will always be like the above (each one indented one level LESS than the one above it). If I see this:

```
    }
    }
```

I know something is wrong. Or if the close curly braces skip a level, I know that something is wrong:

```
if (...) {
    if (...) {
        if (...) {
            statement;
            statement;
        }
}
```

I can tell just by looking at the position of those two close curly braces that something is wrong.

I don't expect this to all make sense right now. The main thing is for you to know that you should be able to track this easily, and with practice you'll be able to.

Section 3.2: The if-else Statement

Now we're ready for our next challenge. After a trial period we have concluded that our $100 bonus policy isn't working that well. We've decided to go with a standard overtime payment plan. In other words, each employee will be paid at one and a half times her normal rate of pay for any hours worked in excess of 40. Here's our first attempt:

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

int main()
{
    int hours;
    int payRate;
    int paycheck;

    cout << "enter hours worked: ";
    cin >> hours;
    cout << "enter rate of pay: ";
    cin >> payRate;

    if (hours <= 40) {
        paycheck = hours * payRate;
    }

    if (hours > 40) {
        paycheck = 40 * payRate + (hours-40) * payRate * 1.5;
    }

    cout << "the amount of the paycheck is "
         << paycheck << " dollars." << endl;
}
```

This solution works fine, but it would be nice if we could simplify it a bit. The situation here is that we have 2 alternative statements, and we always want to execute exactly one of them. We never want to execute none of them, and we never want to execute both of them. C++ provides a statement especially for this kind of situation, since it happens a lot. It's called the if-else statement and it goes like this:

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

int main()
{
    int hours;
    int payRate;
    int paycheck;

    cout << "enter hours worked: ";
    cin >> hours;
    cout << "enter rate of pay: ";
    cin >> payRate;

    if (hours <= 40) {
        paycheck = hours * payRate;
    } else {
        paycheck = 40 * payRate + (hours-40) * payRate * 1.5;
    }

    cout << "the amount of the paycheck is "
         << paycheck << " dollars." << endl;
}
```

The general form of the if-else statement is as follows:

```
if (<condition>){
    <statements1>
} else {
    <statements2>
}
```

If the condition is true, then `statements1`gets executed. If the condition is false, `statements2` gets executed. We call `statements1` the **if-part** and `statements2` the **else-part**.

Section 3.3: Nested if Statements

Since the if-else statement in C++ is itself a statement, it is possible to have if-else statements inside of if-else statements. When we do this it is called **nested** statements. For example, let's say that we live in a country where senior citizens are taxed at a different rate. Here are the tax tables for that country:

	under 55	55 or over
< $20,000	15%	10%
>= $20,000	20%	15%

Here is a program segment that will compute the tax for a citizen of this country:

```
if (age < 55) {

    if (salary < 20000){
        tax = salary * 0.15;
    } else {
        tax = salary * 0.20;
    }

} else {

    if (salary < 20000){
        tax = salary * 0.10;
    } else {
        tax = salary * 0.15;
    }

}
```

Let's follow the execution of this program segment through with a specific example. Let's say that age is 60 and salary is $22,000.

- First we test the condition `age < 55`. Is the condition true or false? It is false, so we skip the if-part and execute the else-part.
- The first thing to do in the else-clause is to test the condition `salary < 20000`. It is also false, so we skip the if-part of that if-else statement and execute the else-part, which says `tax = salary * 0.15`. So `tax` becomes 3300.

Let's try a more complex example. The following program segment will print an "A" if the score is greater than or equal to 90, a "B" if the score is between 80 and 89, a "C" if the score is between 70 and 79, a "D" if the score is between 60 and 69, and an "F" if the score is below 60.

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

int main()
{
    int score;

    cout << "Enter a score: ";
    cin >> score;

    if (score >= 90) {
        cout << "A" << endl;
    } else {
        if (score >= 80) {
            cout << "B" << endl;
        } else {
            if (score >= 70) {
                cout << "C" << endl;
            } else {
                if (score >= 60) {
                    cout << "D" << endl;
                } else {
                    cout << "F" << endl;
                }
```

```
            }
        }
    }
}
```

If score is greater than 90 the first if-part is executed and the else-part is skipped. This means that even though the other four conditions are true, only the `cout << "A"` will be executed, because the other three if-else statements are part of the original else-clause, which is skipped.

Although this program segment works correctly, it is very difficult to read. In a situation like this, where you have several alternatives and exactly one of them is to be executed, you should structure your nested if-else statements like this:

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

int main()
{
    int score;

    cout << "Enter a score: ";
    cin >> score;

    if (score >= 90) {
        cout << "A" << endl;
    } else if (score >= 80) {
        cout << "B" << endl;
    } else if (score >= 70) {
        cout << "C" << endl;
    } else if (score >= 60) {
        cout << "D" << endl;
    } else {
        cout << "F" << endl;
    }
}
```

I think you'll agree that this method is a lot easier to read than the previous example was. When you encounter a complex if-else situation like this, you don't really even have to think too much about all the ifs and elses. Just follow the pattern and realize that exactly one of the alternatives will be executed.

Even though all of our examples so far have had only one statement in each if-clause or else-clause, it is perfectly acceptable to have more than one. For example, if, in our grade example, we wanted not only to print out the grade but to assign the appropriate grade to a character variable called `grade`, our program segment would look like this:

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

int main()
{
    int score;
    char grade;

    cout << "Enter a score: ";
    cin >> score;

    if (score >= 90) {
        cout << "A" << endl;
        grade = 'A';
    } else if (score >= 80) {
        cout << "B" << endl;
        grade = 'B';
    } else if (score >= 70) {
        cout << "C" << endl;
        grade = 'C';
    } else if (score >= 60) {
        cout << "D" << endl;
        grade = 'D';
    } else {
        cout << "F" << endl;
        grade = 'F';
    }
```

```
        //program continues...
    }
```

Section 3.4: Simple Logical Expressions

We need to talk some more about the <condition> part of the if statement. In our first example (section 3.1), the condition was

```
hours > 30.
```

The condition `hours > 30` must be either true or false. If hours is greater than 30, then the condition is true. If hours is not greater than 30, then the condition is false. A **condition** is something that is always either true or false. What I am calling a condition here is more formally known as a **boolean expression** or **logical expression**.

Although you will see exceptions to this rule later on, for now you should assume that every condition must involve one of the following six operators (these are **relational operators**):

Algebra	English	>C++	Example	Result
>	greater than	>	40 > 30	true
<	less than	<	40 < 30	false
≤	less than or equal to	<=	40 <= 30	false
≥	greater than or equal to	>=	40 >= 30	true
=	equal to	==	40 == 30	false
≠	not equal to	!=	40 != 30	true

Notes about the table above:

- Greater-than and less-than look just like they do in algebra.
- Greater-than-or-equal-to, less-than-or-equal-to, and not-equal-to all look a bit different from how they look in algebra. They each require two characters. In each case, the two characters must be typed in the correct order (you can't say =< instead of <=). There cannot be a space between the two characters (you must type !=, not ! =).
- Equal-to requires the use of two equal signs, not just one as in algebra.

One of the most common mistakes that C++ beginners make is to use = when they really mean ==. Make sure that you understand the difference.

differences between = and ==

difference #1:

= is the assignment operator.
== is the "is equal to" operator.

difference #2:

When you use =, you are telling the computer to do something. For example,

`hours = 30;` means "put the value 30 into the variable hours."

When you use ==, you are asking the computer a question. For example,

`hours == 30` means "is the value of hours equal to 30?"

difference #3:

The == is only used in logical expressions. The = is only used in assignment statements.

Unfortunately, if you mess up and put = in your condition instead of ==, C++ will not tell you. It will think you mean something different. Consider this example:

```
hours = 1;
if (hours = 30) {
    cout << "hours is 30";
}
```

In this case, when C++ sees the condition hours = 30, instead of checking to see whether hours is equal to 30, it will do what you have told it to do: put the value 30 into the variable hours. Then it will go ahead and execute the `cout` statement. In summary remember that:

In assignment statements use =
In logical expressions use ==

Section 3.5: Complex Logical Expressions

In C++ we can also use words like "and", "or", and "not" in our logical expressions. They are called **logical operators**. We use special symbols to represent them. For "and" we use the symbol "`&&`" (double ampersand), for "or" we use the symbol "`||`" (double vertical bar), and for "not" we use the symbol "`!`" (exclamation point). The "|" (vertical bar) is sometimes hard to find on the keyboard. It is the <shift> backslash key, right above the return key.

Example 1:

Problem: Write a logical expression that means "`number` is between 5 and 15."

Solution:

```
(number > 5) && (number < 15)
```

This expression should be read "number is greater than 5 and number is less than 15."

Example 2:

Problem: Write a logical expression that means "`number` is not between 5 and 15."

Solution: There are two approaches to this problem. The most obvious approach is to enclose the solution to example 1 in parentheses and then put a "`!`" in front:

```
!((number > 5) && (number < 15))
```

This condition would be read "**not** [number is greater than 5 and number is less than 15]." In other words, `number` is **not** between 5 and 15.

A better solution would be to use "or". In order for `number` to **not** be between 5 and 15, it must be either less than 5 or greater than 15!

```
(number < 5) || (number > 15)
```

Notice that **when you use "and" and "or," you must always have a valid logical expression on both sides of the operator**. In other words,

```
number < 5 || > 15
```

is not a valid way to express the condition from example 2. The reason it is not valid is that, although there is a logical expression on the left side of the "||", there is not a logical expression on the right side. When you use "not," it must be followed by a condition, as it is in example 2.

It is important to be able to evaluate logical expressions even if they become quite complex. Many bugs are introduced into programs as a result of the programmer's inability to evaluate logical expressions correctly. As you know, when you evaluate a logical expression it always turns out to be either "true" or "false". The way to evaluate an expression that has "and", "or", or "not" in it is to first evaluate each of the more simple expressions, and then apply the following rules:

Expression	**Evaluates to**
false && false	false
false && true	false
true && false	false
true && true	true
false || false	false
false || true	true
true || false	true
true || true	true
! true	false
!false	true

This table of rules can be summarized as follows: When the operator is "and," the expression evaluates to "false" if either one of the operands is "false." When the operator is "or," the expression evaluates to "true" if either one of the operands is "true." "Not" in front of an expression simply reverses its value.

One last thing you need to know about evaluating complex logical expressions before we do a few examples. Between the logical operators, "not" has the highest precedence (is evaluated first), "and" comes next, and "or" is last.

Example 1: Evaluate this logical expression:

`(3 < 7) || !(4 < 8)`	
`true || !true`	
`true || false`	(evaluate "not" first)
`true`	

Example 2: Evaluate this logical expression:

`!((3 < 7) || (4 < 8))`	
`!(true || true)`	

`!(true)`	(evaluate inside parentheses first)
`false`	

Example 3: Evaluate this condition:

`(5 < 9) || !(3 < 7) && (4 > 8)`	
`true || !(true) && false`	
`true || false && false`	(evaluate "not" first)
`true || false`	(now evaluate "and")
`true`	

Notice in example 3 that if we had evaluated "or" before evaluating "and" we would have gotten an answer of "false" instead of "true."

In addition to the topics covered in this lesson you should also read about the switch statement in the text.