2.6 — Boolean values and an introduction to if statements

BY ALEX ON JUNE 9TH, 2007 | LAST MODIFIED BY ALEX ON JUNE 21ST, 2018

In real-life, it's common to ask or be asked questions that can be answered with "yes" or "no". "Is an apple a fruit?" Yes. "Do you like asparagus?" No.

Now consider a similar statement: "Apples are a fruit". Is this statement true or false? It's clearly true. Or how about, "I like asparagus". Absolutely false (yuck!).

These kinds of sentences that have only two possible outcomes: yes/true, or no/false are so common, that many programming languages include a special type for dealing with them. That type is called a **boolean** type.

Boolean variables

Boolean variables are variables that can have only two possible values: true (1), and false (0).

To declare a boolean variable, we use the keyword **bool**.

```
bool b;
```

To initialize or assign a true or false value to a boolean variable, we use the keywords **true** and **false**.

```
bool b1 = true; // copy initialization
bool b2(false); // direct initialization
bool b3 { true }; // uniform initialization (C++11)
b3 = false; // assignment
```

Just as the unary minus operator (-) can be used to make an integer negative, the logical NOT operator (!) can be used to flip a boolean value from true to false, or false to true:

```
bool b1 = !true; // b1 will have the value false
bool b2(!false); // b2 will have the value true
```

Boolean values are not actually stored in boolean variables as the words "true" or "false". Instead, they are stored as integers: true becomes the integer 1, and false becomes the integer 0. Similarly, when boolean values are evaluated, they don't actually evaluate to "true" or "false". They evaluate to the integers 0 (false) or 1 (true).

Consequently, when we print boolean values with std::cout, std::cout prints 0 for false, and 1 for true:

```
#include <iostream>

int main()
{
    std::cout << true << std::endl; // true evaluates to 1
    std::cout << !true << std::endl; // !true evaluates to 0

    bool b(false);
    std::cout << b << std::endl; // b is false, which evaluates to 0
    std::cout << !b << std::endl; // !b is true, which evaluates to 1
    return 0;
}
```

Outputs:

```
1
0
0
1
```

If you want std::cout to print "true" or "false" instead of 0 or 1, you can use std::boolalpha:

```
#include <iostream>
```

```
int main()
{
    std::cout << true << std::endl;
    std::cout << false << std::endl;

    std::cout << std::boolalpha; // print bools as true or false

    std::cout << true << std::endl;
    std::cout << false << std::endl;
    return 0;
}
```

This prints:

```
1
0
true
false
```

You can use std::noboolalpha to turn it back off.

A first look at booleans in if-statements

One of the most common uses for boolean variables is inside *if* statements. If statements typically take the following form:

```
if (expression) statement1;
```

or

```
if (expression) statement1;
else statement2;
```

When used in the context of an if statement, the expression is sometimes called a **condition** or **conditional expression**.

In both forms of the if statement, *expression* is evaluated. If *expression* evaluates to a non-zero value, then *statement1* is executed. In the second form, if *expression* evaluates to a zero value, then *statement2* is executed instead.

Remember that true evaluates to 1 (which is a non-zero value) and false evaluates to 0.

Here’s a simple example:

```
if (true) // true is our conditional expression
    std::cout << "The condition is true" << std::endl;
else
    std::cout << "The condition is false" << std::endl;
```

prints:

```
The condition is true
```

Let’s examine how this works. First, we evaluate the conditional part of the if statement. The expression “true” evaluates to value 1, which is a non-zero value, so the statement attached to the if statement executes.

The following program works similarly:

```
bool b(false);
if (b)
    std::cout << "b is true" << std::endl;
else
    std::cout << "b is false" << std::endl;
```

prints:

```
b is false
```

In the above program, when the condition evaluates, variable b evaluates to its value, which in this case is false. False evaluates to value 0. Consequently, the statement connected to the if statement does not execute, but the else statement does.

Executing multiple statements

In a basic form of an *if statement* presented above, statement1 and statement2 may only be a single statement. However, it's also possible to execute multiple statements instead by placing those statements inside curly braces ({}). This is called a block (or compound statement). We cover blocks in more detail in lesson **4.1 -- Blocks (compound statements)**.

An if or if-else using multiple statements takes the form:

```
if (expression)
{
    statement1a;
    statement1b;
    //  etc
}
```

or

```
if (expression)
{
    statement1a;
    statement1b;
    // etc
}
else
{
    statement2a;
    statement2b;
    // etc
}
```

For example:

```cpp
if (true) // when this if statement executes
{ // this block of statements will execute
    std::cout << "The condition is true" << std::endl;
    std::cout << "And that's all, folks!" << std::endl;
}
else
    std::cout << "The condition is false" << std::endl;
```

This prints:

```
The condition is true
And that's all, folks!
```

A slightly more complicated example

The equality operator (==) is used to test whether two integer values are equal. Operator== returns true if the operands are equal, and false if they are not.

```cpp
#include <iostream>

int main()
```

```cpp
{
    std::cout << "Enter an integer: ";
    int x;
    std::cin >> x;

    if (x == 0)
        std::cout << "The value is zero" << std::endl;
    else
        std::cout << "The value is non-zero" << std::endl;
    return 0;
}
```

Here's output from one run of this program:

```
Enter an integer: 4
The value is non-zero
```

Let's examine how this works. First, the user enters an integer value. Next, we use operator== to test whether the entered value is equal to the integer 0. In this example, 4 is not equal to 0, so operator== evaluates to the value false. This causes the else part of the if statement to execute, producing the output "The value is non-zero".

Boolean return values

Boolean values are often used as the return values for functions that check whether something is true or not. Such functions are typically named starting with the word *is* (e.g. isEqual) or *has* (e.g. hasCommonDivisor).

Consider the following example, which is quite similar to the above:

```cpp
#include <iostream>

// returns true if x and y are equal, false otherwise
bool isEqual(int x, int y)
{
    return (x == y); // operator== returns true if x equals y, and false otherwise
}

int main()
{
    std::cout << "Enter an integer: ";
    int x;
    std::cin >> x;

    std::cout << "Enter another integer: ";
    int y;
    std::cin >> y;

    if (isEqual(x, y))
        std::cout << x << " and " << y << " are equal" << std::endl;
    else
        std::cout << x << " and " << y << " are not equal" << std::endl;

    return 0;
}
```

Here's output from one run of this program:

```
Enter an integer: 5
Enter another integer: 5
5 and 5 are equal
```

How does this work? First we read in integer values for x and y. Next, the conditional expression "isEqual(x, y)" is evaluated. This results in a function call to isEqual(5, 5). Inside that function, 5 == 5 is evaluated, producing the value true (since 5 is equal to 5). That value is returned back to the caller. Since the conditional evaluated to true, the statement attached to the if executes, producing the output "5 and 5 are equal".

Boolean values take a little bit of getting used to, but once you get your mind wrapped around them, they're quite refreshing in their simplicity!

Non-boolean conditionals

In all of the examples above, our conditionals have been either boolean values (true or false), boolean variables, or functions that return a boolean value. What happens if your conditional is an expression that does not evaluate to a boolean value?

You already know the answer: If the conditional evaluates to a non-zero value, then the statement associated with the if statement executes.

Therefore, if we do something like this:

```
if (4) // non-nonsensical, but for the sake of example...
    std::cout << "hi";
else
    std::cout << "bye";
```

This will print "hi", since 4 is a non-zero value.

Inputting boolean values

Inputting boolean values using std::cin sometimes trips new programmers up.

Consider the following program:

```
int main()
{
    bool b; // uninitialized variable
    std::cout << "Enter a boolean value: ";
    std::cin >> b;
        std::cout << "You entered: " << b;

    return 0;
}
```

```
Enter a boolean value: true
You entered: 119
```

Wait, what?

It turns out that std::cin only accepts two inputs for boolean variables: 0 and 1 (*not* true or false). Any other inputs will cause std::cin to silently fail. In this case, because we entered "true", std::cin silently failed and didn't assign a value to b. Consequently, when std::cout printed a value for b, it printed whatever uninitialized value was in variable b. Garbage!

Quiz

1) A prime number is a whole number greater than 1 that can only be divided evenly by 1 and itself. Write a program that asks the user to enter a single digit integer. If the user enters a single digit that is prime (2, 3, 5, or 7), print "The digit is prime". Otherwise, print "The digit is not prime".

Hint: Use if statements to compare the number the user entered to the prime numbers. Use a boolean to keep track of whether the user entered a prime number or not.

Quiz answers

1) **Show Solution**

2.7 -- Chars