4.9 — Boolean values

BY ALEX ON JUNE 9TH, 2007 | LAST MODIFIED BY ALEX ON OCTOBER 11TH, 2019

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 that can be answered with a "true" or "false": "Apples are a fruit". 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 (note: Boolean is properly capitalized in the English language because it's named after its inventor, George Boole).

Boolean variables

Boolean variables are variables that can have only two possible values: *true*, and *false*.

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 };
bool b2 { false };
b1 = false;
bool b3 {}; // default initialize to false
```

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 be initialized with the value false
bool b2 { !false }; // b2 will be initialized with 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). Because Booleans actually store integers, they are considered an integral type.

Printing Boolean variables

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*. Here’s an example:

```
#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.

Integer to Boolean conversion

You can’t initialize a Boolean with an integer using uniform initialization:

```
#include <iostream>

int main()
{
    bool b{ 4 }; // error: narrowing conversions disallowed
    std::cout << b;

    return 0;
}
```

(note: some versions of g++ don’t enforce this properly)

However, in any context where an integer can be converted to a Boolean , the integer *0* is converted to *false*, and any other integer is converted to *true*.

```
#include <iostream>

int main()
{
    std::cout << std::boolalpha; // print bools as true or false

    bool b1 = 4 ; // copy initialization allows implicit conversion from int to bool
```

```
    std::cout << b1 << '\n';

    bool b2 = 0 ; // copy initialization allows implicit conversion from int to bool
    std::cout << b2 << '\n';


    return 0;
}
```

This prints:

```
true
false
```

Inputting Boolean values

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

Consider the following program:

```
int main()
{
    bool b {}; // default initialize to false (0)
    std::cout << "Enter a boolean value: ";
    std::cin >> b;
        std::cout << "You entered: " << b;

    return 0;
}
```

```
Enter a Boolean value: true
You entered: 0
```

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. In C++11 or newer, this also sets *b* to 0 (which is the same value it had already been initialized with). Consequently, when *std::cout* printed a value for *b*, it printed 0.

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:

```
#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()
{
```

```
11      std::cout << "Enter an integer: ";
12      int x{ 0 };
13      std::cin >> x;
14
15      std::cout << "Enter another integer: ";
16      int y{ 0 };
17      std::cin >> y;
18
19      std::cout << std::boolalpha; // print bools as true or false
20
21      std::cout << x << " and " << y << " are equal? ";
22      std::cout << isEqual(x, y); // will return true or false
23
24      return 0;
25  }
```

Here's output from two runs of this program:

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


Enter an integer: 6
Enter another integer: 4
6 and 4 are equal? false
```

How does this work? First we read in integer values for *x* and *y*. Next, the expression "isEqual(x, y)" is evaluated. In the first run, this results in a function call to isEqual(5, 5). Inside that function, 5 == 5 is evaluated, producing the value *true*. The value *true* is returned back to the caller to be printed by std::cout. In the second run, the call to isEqual(6, 4) returns the value *false*.

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! Boolean values are also a huge part of the language -- you'll end up using them more than all the other fundamental types put together!

We'll continue our exploration of Boolean values in the next lesson.


4.10 -- Introduction to if statements


Index


4.8 -- Floating point numbers

C++ TUTORIAL | PRINT THIS POST

536 comments to 4.9 — Boolean values

« Older Comments 1 ... 6 7 8

Scarlet Johnson
February 8, 2020 at 10:13 pm · Reply

```
#include <iostream>
#include <iomanip>
int main()
{
    double a{0.1};
    std::setprecision(17);
    std::cout << a;


    return 0;
}
```

This program on running shows 0.1 with no precision. Explain?

nascardriver
February 9, 2020 at 7:07 am · Reply

`std::setprecision` doesn't do anything on its own. It returns an object that can be passed to a stream to modify the stream's precision.

```
std::cout << std::setprecision(17);
```

Scarlet Johnson
February 9, 2020 at 7:53 am · Reply

Oook! Thanks.

« Older Comments 1 ... 6 7 8