Boolean Variables and Logical Expressions

A boolean variable is a variable that can take on one of only two values—`true` or `false`

A *logical expression* (also known as a *condition*) is an expression that can only have two possible outcomes—`true` or `false`.

You will also sometimes hear these values or outcomes referred to as

1. **on** or **off**
2. **1** or **0**
3. **high** or **low**

The following example demonstrates boolean variables and some simple operations on them:

```
/***** boolean variables & expressions ********

  This is not a program in the accepted sense.
  Rather it is a series of isolated examples
  strung together as if they were a program.

**********************************************/



int main(){
    int n = 3;

    bool isOn = true;
    bool flag;
    bool toggle = false;

    flag = !isOn;
    toggle = !toggle;

    flag = n;
    toggle = !toggle;

    flag = (n != 0);
    toggle = !toggle;

    flag = n < 3;
    toggle = !toggle;

    flag = n >= 3 && n < 6;
    toggle = !toggle;

    return 0;
}
```

The above example introduces the *boolean operator* not (`!`) sometimes called the bang. The line

`flag = !isOn;`

sets `flag` to the opposite of `isOn` (`true`), so `flag` becomes `false`.

The line

`toggle = !toggle;`

implements a simple toggle switch which changes state every time the line is executed.

There are three *boolean operators*, that is operators whose operand(s) are boolean.

Operator	Kind	Relationship
&&	binary	And
||	binary	Or
!	unary	Not

This table defines the boolean operators

a	b	a && b	a || b	!a
false	false	false	false	true
false	true	false	true	true
true	false	false	true	false
true	true	true	true	false

`a||b` is `true` if *either* `a` or `b` is `true`

`a&&b` is `true` only if *both* `a` and `b` are `true`

Logical Expressions

Lines like

`flag = n < 3;`

are a little more complicated. `n` is an integer (which happens to have the value `3`).

`n < 3` is a *logical expression*, that is `n` is either less than `3` or it isn't.

The *relational operator* less than (<) has a lower precedence than = so the expression is evaluated first (to `false` since 3 is *not* less than 3) and then `flag` is set to `false`.

There are six relational operators, all of which take two operands.

operator	operation
==	*left operand* **is equal to?** *right operand*
!=	*left operand* **is not equal to?** *right operand*
<	*left operand* **is less than?** *right operand*
>	*left operand* **is greater than?** *right operand*
<=	*left operand* **is less thanor equal to?** *right operand*
>=	*left operand* **is greater thanor equal to?** *right operand*

The outcome of a relational operation is always boolean—`true` or `false`.

bool_2.cpp

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

/***** complex boolean expressions ********

  In this example we examine complex bool
  expressions. In particular we look at
  the SHORT-CIRCUIT property.

**********************************************/



int main(){
    bool flag;
    int number;

    cout << "Please input an integer: ";
    cin >> number;

    // flag = a + b.c

    flag = number == 4 || number >= 7 && number <= 10;

    cout << "The flag is " << flag <<'\n';
    return 0;
}
```

This example is more complex. It contains the following logical expression:

```
number == 4 || number >= 7 && number <= 10
```

which combines logical and boolean operators.

The logical operators have lower precedence than the relational ones so

1. the relational operators are evaluated giving `bool` results
2. the bool results are then combined (&& before ||)

so the expression above reads

(`number` isEqualTo? 4) **or**
(`number` isGreaterThanOrEqualTo? 7) **and** (`number` isLessThanOrEqualTo? 10)

Logicists would consider the three phrases to be *propositions* each of which is either `true` of `false`

As the comment in the example shows, in boolean algebra, the expression

a **or** b **and** c is also written a**+**b**.**c where the '+' stands for or and the . or x for and. The mathematical operators make the precedence of **and** over **or** clear.

Table of Precedence

Here is a table for the precedence of all the operators we know about so far

Operator	Precedence	Description
! + -	Highest	logical not, unary plus, unary minus
* / %		multiplication, division, modulo
+ -		addition, subtraction
< <= > >=		relational inequalities
== !=		equal, not equal
&&		and
||		or
=	Lowest	assignment

Bool Conversions

Conversion to Bool

Integer types are converted to `bool` as follows:

1. `0` is converted to `false`
2. anything else is converted to `true`

Integer types include both `int` and `char`. Note also that since `doubles` can be converted to `ints`, this effectively means `doubles` can be converted to `bool` as well.

If an `int` value occurs where a `bool` is expected, this conversion is often applied automatically, e.g

```
if (i) cout << "i is not 0";
else cout << "i is 0";
```

This is generally regarded as poor style.

Note: We haven't actually studied `if` yet (next topic!) but the meaning should be clear.

Conversion from Bool

values of type `bool` can be converted to `int` as follows:

1. `false` is converted to `0`
2. `true` is converted to `1`

Again, if a `bool` value is encountered where an `int` is expected the conversion can occur automatically. In the following example `flag` is a `bool` and `x` is a `double`:

```
x = x + flag;
```

Since `x` is a `double` and only a `double` can be added to a `double`, the value of `flag` is first converted to an `int` (`0` or `1`) and then that `int` is converted to a `double` (`0.0` or `1.0`).

If `flag` is `false`, `x` remains unchanged. If `true`, `1.0` is added to `x`.

Such "clever" programming is seldom justified and we will penalize it as bad style.

The Short Circuit Property

C++ (and many languages which borrow its syntax such as Java, JavaScript, PHP) have something know as the *short- circuit property*.

Within the bounds of precedence, boolean expressions are executed left to right. Once the outcome of the expression is known, execution stops with no farther evaluation.

In the above example, the steps are as follows.

1. evaluate whether the first proposition is true or false (`number==4`)
2. evaluate the second proposition (`number >=7`)

3. evaluate the third proposition (`number <= 10`)
4. combine the second and third results by **and**ing them
5. combine first result by **or**ing it with result of step 4.

if we input to number a value of 4, the first proposition will be `true`. Since that guarantees the entire combined proposition is `true` (`true` **or** anything else is always `true`), step 1 is the only step that is executed.

If we input 5 or 6, the first proposition will be `false`, proposition two will be `false`, guaranteeing the entire proposition is `false`, so execution stops after step 3.

Even or Odd?

Consider the following example:

bool_3.cpp

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

/***** odd or even numbers ********

  In this example we utilize the properties
  of integer arithmetic to determine if a
  number is odd or even.

**********************************************/



int main(){
    bool even;
    int number;

    cout << "Please input an integer: ";
    cin >> number;

    even = (2*(number/2) == number);

    cout << "The number is ";
    if (even)
            cout << "even.\n";
    else
            cout << "odd.\n";

    return 0;
}
```

our logical expression

`( 2 *(number/ 2 ) == number)`

has only got one logical operator. The rest of it involves a little computer trickery.

It uses the special properties of integer division to check if number is even

When we divide an even number by 2 there will be no remainder so when we multiply by 2 again we get the original number back.

Dividing an odd number by 2, however gives us a fractional part which is discarded (integers can't hold fractional parts). Thus when we multiply by 2 again we don't get the original number back.

Again, the progam uses an `if` statement which poaches on our next topic. But again, the intent should be pretty clear.

Most C++ programmers would see the following as improved. Why?

bool_4.cpp

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

/***** odd or even numbers ********

  A little better version of using
  integer arithmetic to determine if a
  number is odd or even.

**********************************************/


int main(){
    int number;

    cout << "Please input an integer: ";
    cin >> number;

    cout << "The number is ";

    if (2*(number/2) == number)
            cout << "even.\n";
    else
            cout << "odd.\n";

    return 0;
}
```

This page last updated on Monday, February 2, 2004