06. Booleans

Objectives

- Use boolean literals
- Use boolean operators
- Declare boolean variables
- Declare and initialize boolean variables
- Use assignment operator for boolean variables
- Declaring constant boolean variables

Boolean values represent “true” and “false”. We'll see later that this type is important for writing programs that can make decisions.

Booleans

An integer can be 5 or 42 or 1024. A double can be 2.141.

A **boolean** value is either true or false. Boolean values are used by programs to make decisions (among other things). Up to this point our programs can now accept input(s) and perform computations one statement at one at a time.

THAT'S NOT GOOD ENOUGH!

Programs should be smart enough to decide which statement to execute. If you're playing a game, you want the game to decide when to restart a game right? You do want to gain extra points when you hit a rogue spaceship, correct??? A program that analyzes the image file of a brain scan should only print “THERE'S BRAIN TUMOR!” only when there's really a brain tumor right?!?

All these and more require **decision-making**. You want to write a program that says (more or less):

```
if rocket hits bad ship then
        bad ship explodes
        player gets 10 points
```

The condition

```
rocket hits bad ship
```

is of course either true or false. That's the point of having booleans.

Another use of booleans is to decide/control when a **loop** or **repetition** should end. For instance, look at this “program” from a DVD player software:

```
“continue playing the video stream and audio stream
as long as the STOP button is not pressed”
```

or to make it more like a “program”:

```
while STOP button is not pressed:
        play the next 0.01 second of video stream and the
        next 0.01 second of audio stream
```

The condition “STOP button is not pressed” again is either true or false. It has a boolean value.

So … before we can talk about decision-making and (also known as branching) and loops/repetition, you need to know how to work with booleans.

Whereas `int`s and `double`s and `float`s can have lots of values, note that booleans have only **two**: `true` and `false`.

Exercise. Run this program.

```
std::cout << true << ' '
          << false << std::endl;
```

What does C++ display `true` as? _____

What does C++ display `false` as? _____

Boolean operators: and

Booleans, like `int`s and `double`s, have operators. Just like + allows you to combine two `int`s into one:

```
1 + 2
```

(i.e. `3`) boolean operators take boolean value(s) and return a boolean value. Why do you need them? Here's a simple example:

(I'm learning C++) and (I'm a student at CC)

Is this true or false? Well this becomes:

(true) and (true)

(Otherwise why are you here???) Of course you know that this is

true

So what I'm saying is that the **"and"** is a **binary operator** that takes (true) and (true) and evaluates to (true). In a C++ program "and" is written

&&

Too bad that's something you have to remember. `&&` is a **binary** boolean operator since you need to feed it **two** boolean values to get a result. (Instead of calling it the "and" operator, mathematicians/computer scientists call it the "logical and" operator.)

To define "and" you need to figure out all the possible cases:

```
true && true     = true
true && false    = false
false && true    = false
false && false   = false
```

This is called a **truth table** of "and" or `&&`. Make sure you can reproduce the above truth table!!!

Exercise. Verify the result of the && operator by completing this program. Complete and run this program and make sure C++ is not crazy.

```
std::cout << "true && true  = " << (true && true) << '\n'
          << "true && false = " << (true && false) << '\n'
          <<
          <<                                                ;
```

Exercise. By the way, what happens if you remove the parentheses?, i.e., what happens when you do this:

```
std::cout << true && true << '\n';
```

Exercise. What is the value of this boolean expression

```
false && ________
```

even though the second value is missing? This is the same for

```
________ && false
```

Boolean operator: or

Another important boolean operator is “or” (or “logical or”). Here's an example:

(I'm learning C++) or (I have 3 arms)

Assuming you do not have three arms, the resulting boolean value is of course:

(true) or (false) = (true)

In C++ “or” is written

| |

(two vertical broken lines – look at the left of the keyboard, above the backslash \ key on your keyboard).

Exercise. Of course we must have a truth table for “or”: Verify this table using a C++ program:

true || true = true
true || false = true
false || true = true
false || false = false

```
std::cout << "true || true = " << (true || true) << '\n'
          <<
          <<
          <<                                                ;
```

Same thing again ... make sure you can reproduce the above table.

Exercise. What is the value of this boolean expression

```
true || _________
```

even though the second value is missing? What about this:

```
_________ || true
```

At this point we have two boolean operators. What if we have a boolean expression like this:

```
false && false || true
```

Note that precedence is important since

```
  (false && false) || true
= false || true
= true
```

and

```
  false && (false || true)
= false && true
= false
```

which gives us different results!!!

The precedence rule is simple:

()	highest priority
&&	
`||`	lowest priority

and as usual for (boolean) operators of the same priority level, you perform left-to-right.

Let's evaluate this by hand:

```
  true || false && (false || true) && false
= true || false && (true) && false by false || true = true
= true || false && true && false   by (true) = true
= true || false && false           by false && true =false
= true || false                    by false && false =false
= true                             by true && false = false
```

Now verify it with C++:

```
std::cout << (true || false && (false || true) && false)
          << std::endl;
```

You should get 1 which means that boolean value is `true`.

Exercise. First evaluate this by hand:

```
(true || false) && false || true && false
```

Next verify it with C++:

Exercise. First evaluate this by hand:

```
(false && false) || (false || true) && false
```

Next verify it with C++:

Exercise. Check that && and || are commutative, i.e. suppose x and y are boolean values, then

x && y is the same as y && x
x || y is the same as y || x

(Here's a memory aid: “I have three arms and I am the Queen of England” has the same boolean value as “I am the Queen of England and I have three arms”. That's pretty obvious, right?)

Exercise. Check that if x is a boolean value, then

```
x && x  is the same as  x
x || x  is the same as  x
```

(Here's a memory aid: “I have a space shuttle in my backyard and I have a space shuttle in my backyard” is the same as “I have a space shuttle in my backyard”. Also, “I have ten dollars or I have ten dollars” is the same as “I have ten dollars”. Right?)

Boolean operator: not

So far we have “and” (i.e. && in C++-speak) and “or” (i.e. `||` in C++-speak).

There's another operator called “not” (or “logical not”). In C++ it's written

!

This operator is **unary** – it requires one value. It simply inverts the boolean value. You write the boolean value on the right-hand side of !. In other words you write

```
!true
```

and not

```
true!
```

Exercise. Complete the following truth table

!true =
!false =

by completing and running this program:

```
std::cout << "!true = " << (!true) << '\n'
          <<                        << std::endl;
```

Exercise. Check that if x is a boolean value then

!!x is the same as x

[Memory aid: “It is not true that I am not tall” is the same as “I am tall”.]

Here the precedence rules for &&, ||, and !:

```
()      highest precedence level
!
&&
||      lowest precedence level
```

(Memory aid: remember unary operators always have higher priority. This is the same for integer and floating point operators.)

Exercise. Evaluate this by hand:

```
(true || false) && !false || true && !false
```

Now verify it with C++:

Exercise. Evaluate this by hand:

```
!true && !(false && !false || true) && false
```

Now verify it with C++.

Other boolean operators

So far the boolean operators take boolean values (true and false) and returns a boolean value.

Now we want to think about those that take numeric values (ints, doubles, floats) and spit out boolean values.

Exercise. Run this program.

```
std::cout << (1 < 2) << std::endl;
std::cout << (1 > 2) << std::endl;
std::cout << (4 < 3) << std::endl;
std::cout << (4 > 3) << std::endl;
std::cout << (5 <= 8) << std::endl;
std::cout << (5 >= 8) << std::endl;
std::cout << (24 == 42) << std::endl;
std::cout << (24 != 42) << std::endl;
```

Of course you know

<	less than
>	greater than

and you can easily guess

<=	less than or equal to
>=	greater than or equal to

It's probably harder to guess that

==	is equal to
!=	is not equal to

By the way <= and >= CANNOT be written as

=<	BAD, BAD, BAD!!!
=>	BAD, BAD, BAD!!!

Exercise. Run this program.

```
std::cout <<  (5 < = 8) << std::endl;
std::cout <<  (5 > = 8) << std::endl;
std::cout <<  (24 = = 42) << std::endl;
std::cout <<  (24 =! 42) << std::endl;
std::cout <<  (24 => 42) << std::endl;
```

What's wrong? Fix it!!!

It's not too surprising that the above boolean operators (<, <=, >, >=, ==, !=) work for doubles too. Yes?

Exercise. Check your instructor was not lying by running this program:

```
std::cout <<  (1.1 < 2.2) << std::endl;
std::cout <<  (1.1 > 2.2) << std::endl;
std::cout <<  (4.4 < 3.3) << std::endl;
std::cout <<  (4.4 > 3.3) << std::endl;
std::cout <<  (5.5 <= 8.8) << std::endl;
std::cout <<  (5.5 >= 8.8) << std::endl;
std::cout <<  (24.24 == 42.42) << std::endl;
std::cout <<  (24.24 != 42.42) << std::endl;
```

Associativity and precedence rules

What happens when you want C++ to evaluate a boolean expression like this:

```
x + y < a * b - c
```

where x, y, a, b and c are (say) integer variables? C++ must of course work on one operator at a time. In general, the numeric operators go first. In other words the +, *, - in the above expression goes before <.

In general the precedence rules now look like this:

*, /, %, +, - (binary and unary)
<, <=, >, >=, !=, ==

The ! Is slightly special. It's unary. So the precedence is pretty high.

Specifically we have

+, - (unary)	highest precedence level
!	
*, /, %	
+, - (binary)	
<, <=, >, >=	
==, !=	
&&	
||	lowest precedence level

By the way = is also an operator – the assignment operator. It's not too surprising that for the following expression:

x = a + b – c

= is the last thing to happen (see previous notes). In fact = is the last in the precedence rules:

+, – (unary)	highest precedence level
!	
*, /, %	
+, – (binary)	
<, <=, >, >=	
==, !=	
&&	
||	
=	lowest precedence level

The associativity rules are what you expect.

a <= b <= c is the same as (a <= b) <= c

(This actually causes a very common error … see next section for a

VERY IMPORTANT gotcha) and

! ! ! (1 == 2) is the same as !(!(!(1 == 2)))

So there's nothing much to remember.

Let's try to compute the boolean value of

```
3 < 2 * 3 && 9 / 2 < 6 * 3
```

Here we go:

```
  3 < 2 + 3 && 9 / 2 < 6 * 3
= 3 < 2 + 3 && 4 < 6 * 3        by 9 / 2 = 4
= 3 < 2 + 3 && 4 < 18           by 6 * 3 = 18
= 3 < 5 && 4 < 18               by 2 + 3 = 5
= true && 4 < 18                by 3 < 5 = true
= true && true                  by 4 < 18
= true                          by true && true
```

Make sure you check this with a program:

```
std::cout << (3 < 2 + 3 && 9 / 2 < 6 * 3) << '\n';
```

Exercise. Evaluate

```
1 + 2 < 2 * 2 && 3 - 1 < 5 - 2 || 5 * 2 < 6 * 3
```

and verify with C++.

Exercise. Evaluate

```
6 * 2 < 2 + 6 && (3 - 1 < 5 - 2 || 5 * 2 < 6 * 3)
```

and verify with C++.

Exercise. Evaluate

```
6 * 2 < 8 && (2 >= 5 - 3 || (2 != 6 * 3 || 3 < 4))
```

and verify with C++.

Gotcha

Here's a **very very very common gotcha** that unfortunately even experienced programmers make. Look at this:

$$-3 <= -2 <= -1$$

This seems to be similar to the following in math:

$$-3 \leq -2 \leq -1$$

and in math

$$-3 \leq -2 \leq -1$$

is true, right? Try this:

```
std::cout << (-3 <= -2 <= -1) << std::endl;
```

The output is 0!!! Which means false!!! Whoa ... what's happening?!? Let's evaluate this the way C++ would::

$$-3 <= -2 <= -1$$

is the same as

$$(-3 <= -2) <= -1$$

Now

$$-3 <= -2 \text{ is true}$$

That means that

$$\begin{aligned} & -3 <= -2 <= -1 \\ = & (-3 <= -2) <= -1 \\ = & \text{true} <= -1 \end{aligned}$$

Here lies the problem. This is a comparison between a boolean and an integer. C++ type cast the boolean true to 1:

$$\begin{aligned} & -3 <= -2 <= -1 \\ = & (-3 <= -2) <= -1 \\ = & \text{true} <= -1 \\ = & 1 <= -1 \end{aligned}$$

and of course 1<= -1 is false

$$\begin{aligned} & -3 <= -2 <= -1 \\ = & (-3 <= -2) <= -1 \\ = & \text{true} <= -1 \\ = & 1 <= -1 \end{aligned}$$

So how do we fix it? Well ***mathematically***

$$-3 \leq -2 \leq -1$$

actually means

"-3 ≤ -2 and -2 ≤ -1"

Try this:

```
std::cout << ((-3 <= -2) && (-2 <= -1)) << '\n';
```

So if you want to (mathematically) say

$$a \leq b \leq c$$

then in C++ you want to say

```
a <= b && b <= c
```

Likewise the C++ expression for the mathematical expression:

$$a < b < c$$

is

```
a < b && b < c
```

Don't say I didn't warn you.

Boolean variables

The C++ name for the boolean type is `bool`.

Declaring boolean variables is easy. Try this program:

```
bool earthIsFlat = false;
std::cout << "earth is flat: " << earthIsFlat
          << std::endl;
```

Exercise. Declare a boolean variable `totallyBroke` and initialize it with the “and” of the all the three variables in the following code segment. Print the values of all the variables.

```
bool noSavings = true;
bool noInheritance = true;
bool noJob = true;
```

Of course you can use an expression that evaluates to a boolean value:

```
int a = 42;
int b = 24;
bool x = (a == b);
std::cout << x << std::endl;
bool y = (b < a && b + 5 < a);
std::cout << y << std::endl;
```

Exercise. Suppose you're given an integer variable `x`. Write down the a boolean expression that is true exactly when `x` is greater than 1 and less than 10. Test your code with `x` = 0, 1, 5, 10, 20.

```
int x;
std::cin >> x;
bool b = ____________________________________;
std::cout << b << std::endl;
```

Exercise. Write a program that prompts the user for the following values:

- amount in the user's savings account
- monthly rent
- daily expenses
- number of days in the current month

(choose appropriate names and types). Next declare a boolean variable `callHomeForCash` and set it to true exactly when the amount in the user's savings account is less than the total expenses for the current month. Print all the values.

Exercise. Verify that you can declare boolean constants.

Boolean input (console window)

You can input boolean values from the keyboard in a console application. However you do ***not*** enter true or false. You enter 1 for true and 0 for false. Try this:

```
bool b;
std::cin >> b;
std::cout << b << std::endl;
```

Exercise. Run your above program again. This time enter integer 5. Next try integer 42. Hmmmm. What's happening?

Exercise. You are commissioned to write a software for Bowing's new plane. The software takes the following as input:

- distance to nearest airport (mile)
- amount of fuel left (gallons)
- mileage (miles per gallon with only one pilot)
- number of passengers (excluding pilot)
- average weight of each passenger (in lbs, excluding pilot)
- thunderstorm in flight path to nearest airport

(choose appropriate names and types). With passengers, the mileage decreases by 1/100 of the total weight of the passengers (excluding the pilot). For instance if the mileage is 1000 miles/gallon, then with an extra passenger weight of 200lbs, the mileage becomes 1000/(200/100). Furthermore, if there is indeed a thunderstorm in the path towards the nearest airport, the mileage drops by a factor of 3. For instance if the mileage (after taking into account the extra passenger weight) is 300 miles per gallon, then with the presence of a thunderstorm in the path, the mileage becomes 300/3. Finally declare a boolean variable `canMakeIt` and set it to true exactly when the plane can arrive at the nearest airport. Write a program to prompts the user for all the above data and print `canMakeIt`.

Some simplification tricks

Here's a summary of some simplification rules for ! where x and y are numeric (either `int`s or `double`s or `float`s):

`!(x == y)`	same as	`x != y`
`!(x != y)`	same as	`x == y`
`!(x < y)`	same as	`x >= y`
`!(x <= y)`	same as	`x > y`
`!(x > y)`	same as	`x <= y`
`!(x >= y)`	same as	`x < y`

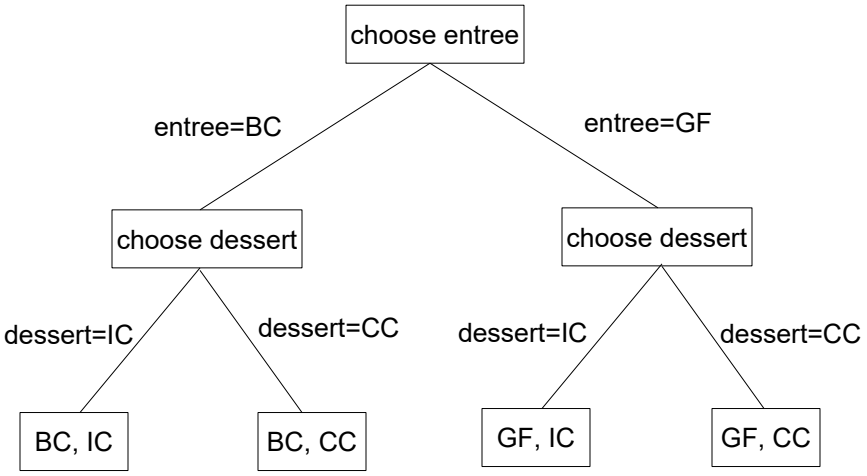
Exercise. Simplify

!(a + b > c + d + e)

Using a C++ program, check that your new expression has the same boolean value as the original expression above.

Decision trees

You must have seen something like this before. Suppose you go to a restaurant and you were told that there are two entrees: barbecue chicken (BC) and grilled fish (GF) and there are two desserts: ice cream (IC) or carrot cake (CC). This is the decision tree of all possible meals:



This diagram lists the consequence of decisions. For instance if you choose BC for entree and IC for dessert, the consequence (see lower left of the above diagram) is that you have a meal of

BC, IC

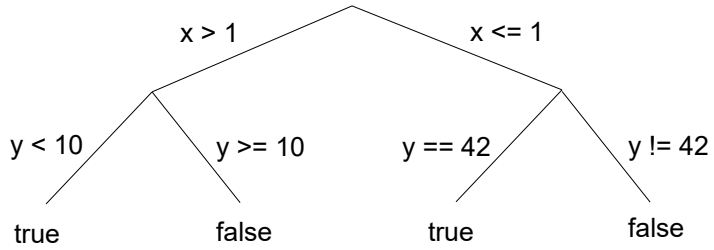
At each point, there are several options. Etc. This is sometimes called a decision tree. Decision trees might also contain information such as cost benefit, risk analysis, etc.

For me, I'll use this to show you what happens when you have a complex boolean expression. Here we go.

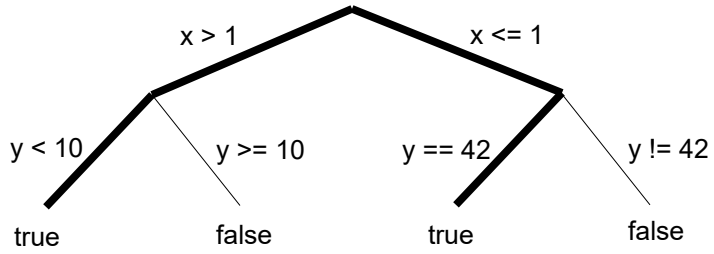
Suppose you're given integer variables `x` and `y`. Write down a boolean expression that is true exactly when the following is true:

- If `x` is greater than 1, then y must be less than 10
- If x is less than or equal to 1, then y must be 42.

Here are all the possible cases:



The boolean expression we want is one that will take us from the top to the bottom with a value of true. There are of course two paths:



i.e., either the take the path where x > 1 and y < 10 or you take the path where x <= 1 and y == 42.

The boolean expression that guarantees a true value is then

```
(x > 1 && y < 10) || (x <= 1 && y == 42)
```

Ignore the boolean expression for now and look at the original requirement:

"Write down the a boolean expression that is true exactly when the following is true:

- If `x` is greater than 1, then y must be less than 10
- If x is less than or equal to 1, then y must be 42."

What will you get if x = 0 and y = 0? Since x = 0, you should look at the second case:

"Write down the a boolean expression that is true exactly when the following is true:

- If `x` is greater than 1, then y must be less than 10
- **If x is less than or equal to 1, then y must be 42.**"

In this case, your boolean expression must be true exactly when y is 42. But our y is 0. So we must have a false value. Now let's check our boolean expression and see if we do get a false value. Substituting x = 0 and y = 0 into:

```
(x > 1 && y < 10) || (x <= 1 && y == 42)
```

we get

```
(0 > 1 && 0 < 10) || (0 <= 1 && 0 == 42)
```

Now let's compute the boolean value of this expression:

```
  (0 > 1 && 0 < 10) || (0 <= 1 && 0 == 42)
= (false &&  true ) || ( true  &&  false )
= (     false     ) || (      false      )
= false
```

Voila! Our boolean expression is correct in this case.
Exercise. Let's check one more case.

(a) From the requirements, what is the expected boolean value when x = 10 and y = 10?
(b) What is the boolean value of our boolean expression

```
(x > 1 && y < 10) || (x <= 1 && y == 42)
```

when x=10 and y=10?
(c) Does (a) match (b)?

Exercise. One more check ...
(a) From the requirements, what is the expected boolean value when x = 1 and y = -5?
(b) What is the boolean value of our boolean expression

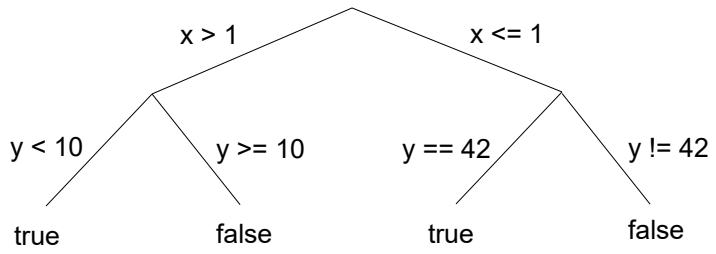
```
(x > 1 && y < 10) || (x <= 1 && y == 42)
```

when x = 1 and y = -5?
(c) Does (a) match (b)?

Exercise. To speed up the verification process you can write a code to evaluate the boolean expression:

```
int x, y;
std::cin >> x >> y;
bool b = (x > 1 && y < 10) || (x <= 1 && y == 42);
std::cout << b << std::endl;
```

When you're testing your boolean expression, you SHOULD test all possible branches of the decision tree:



In other words your test cases must run through all the four possible paths from the top to the bottom. For instance here are 4 cases:

- x = 2, y = 0
- x = 10, y = 10
- x = 0, y = 42
- x = -1, y = 10

In fact, in testing boolean expressions (or testing software in general), you should also test near boundary cases. For instance, in the above, when x > 1, the value y = 10 is the boundary case. So it is advisable for x = 10, one should test the cases y = 9, y = 10, y = 11. For the case when x <= 1, the boundary case is at y = 42. So it makes sense to test y = 41,

42, 43. In fact at the higher level, x has a boundary case of x = 1. So it's also advisable to test x = -1, x = 0, x = 1.

Exercise. Suppose you're given integer variables x and y. Write a boolean expression that is true exactly when the following is true:

- If x is even, then y is less than 10
- If x is odd, then y is greater than 20

Test your boolean expression thoroughly.

Exercise. Suppose you're writing a computer game and you have the following variables.

fuel_level	int	The amount of fuel left. Range from 0 to 1000.
shield_energy	int	Range from 0 to 3.
laser_pod_energy	double	Ranges from 0.0 to 1.0. The player can fire the laser only when this is 1.0, i.e., when the laser pod is fully charged.
num_laser_collides	int	Number of enemy laser that collides with your ship.
num_asteroid_collides	int	Number of asteroids that collides with your ship.

(You of course do not need to know how values are assigned to these variables. Just assume that they have the right values.) Write a boolean expression that tells us if the game should continue. The game should continue exactly when the following is true:

Your fuel level is greater than 0 and furthermore one of the following is true.

- If your shield energy is 0: the number of lasers and asteroids that collides with you is 0.
- If your shield energy is 1: the number of lasers is at most 1 and no asteroid has collided with you
- If your shield energy is 2: the number of lasers is at most 5 and the number of asteroid that is in collision with you is at most 2.
- Your shield energy is 3.

De Morgan's laws

Look at this statement. John said to Mary:

"I'm not stupid and I'm not ugly!!!"

What about this. John said

"It's not true that I'm either stupid or ugly!!!"

Note that they both mean the same thing. In other words if b and c are boolean values, then

```
(!b) && (!c)
```

is the same as

```
!(b || c)
```

Exercise. Complete this program to verify the above formula by initializing `b` and `c` with all possible choices of boolean values.

```
bool b = true, c = true;
std::cout << (!b && !c) << ' ' << !(b || c) << '\n';
```

There is a similar formula:

```
(!b) || (!c)
```

is the same as

```
!(b && c)
```

The two facts:

`(!b) && (!c)`	is the same as	`!(b || c)`
`(!b) || (!c)`	is the same as	`!(b && c)`

are called **de Morgan's laws**.

Why are these formula helpful? Here's one use of de Morgan's Laws to simplify a boolean expression:

!(x < 0 || x > 5) = !(x < 0) && !(x > 5)

Now note that !(x < 0) is the same as x >= 0. Right? Also, !(x > 5) is the same as x <= 5. So

!(x < 0 || x > 5) = !(x < 0) && !(x > 5)
= (x >= 0) && (x <= 5)

Note that the final boolean expression does not have the ! operator. Altogether

!(x < 0 || x > 5)	uses 4 operators
(x >= 0) && (x <= 5)	uses 3 operators

Get it?

Exercise. Using de Morgan's law, simplify this expression:

```
!(x < y + z &&  y - z == x)
```

Write a simple C++ program to verify your simplification.

Exercise. Using de Morgan's law, simplify this expression:

```
!(x < y + z || y != x + y &&  y - z == x)
```

Write a simple C++ program to verify your simplification.

Exercise. Using de Morgan's law, simplify this expression:

```
!((x < y + z || y != x + y) &&  y - z == x)
```

Write a simple C++ program to verify your simplification.

Exercise. Simplify this expression:

```
!((x < y * z || y % 3 < 1) &&  y >= x + 1)
```

(All variables are integer variables with positive values.) Write a simple C++ program to verify your simplification.

An easy simplification

Here's something you should know: If `x` is a boolean variable. Then

```
x
```

and

```
x == true
```

have the same boolean value. That's not too surprising since the following pairs of statements are the same:

“I'm broke.”	“It is true that I'm broke.”
“I'm ten feet tall.”	“It is true that I'm ten feet tall.”
“I have a pet tiger.”	“It is true that I have a pet tiger.”

Right?

Let me try to prove that. There are only two possible values for `x`: it's either `true` of `false`.

CASE: `x` is `true`. In that case

```
x == true
= true == true
= true
```

which is the value of `x`.

CASE: `x` is `false`. In that case

```
x == true
= false == true
= false
```

which is, again, the value of `x`.

So in all cases,

```
x
```

and

```
x == true
```

have the same boolean value.

Similarly

```
!x
```

and

```
x == false
```

have the same boolean value.

Exercise. Here's a boolean expression for “It is true that the earth is flat, it is true that the speed of light is 186000 miles/second, it is true that I have a pet tiger, and it is false that I cannot tie sheloaces.”:

```
(earth_is_flat == true)
&& ((LIGHT_SPEED == 186000) == true)
&& (have_pet_tiger == true)
&& (cannot_tie_shoelaces == false)
```

Simplify it (so that == does not appear in your simplified expression.) Test it with a C++ program.

Summary

The boolean type has two values: `true` and `false`. The C++ name of the boolean type is `bool`.

C++ supports the following operators for boolean values:

`&&`	(logical) and
`||`	(logical) or
`!`	(logical) not

They are defined by the following truth tables:

```
true && true     = true
true && false    = false
false && true    = false
false && false   = false

true || true     = true
true || false    = true
false || true    = true
false || false   = false

!true            = false
!false           = true
```

C++ supports the following binary boolean operators that accept numeric values (integer or floating point types):

<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	equal to
!=	not equal to

The following gives the precedence rules for a boolean expression containing the following operators:

+, - (unary)	highest precedence level
!	
*, /, %	
+, - (binary)	
<, <=, >, >=	
==, !=	
&&	
`||`	lowest precedence level

There is an automatic type casting between numeric types (integers and floating point types) and boolean values. Zero is type casted to false and vice versa. Nonzero values are typecasted to true.

Exercises

Q1. Evaluate the following boolean expression or indicate the error:

1 + 2 < 2 * 2 && 3 – 1 < 4 – 2 || 5 * 2 < 6 * 3

Q2. Evaluate the following boolean expression or indicate the error:

5 + 2 < 2 * 1 && (3 / 2 < 4 % 2 || 3 * 2 < 6 % 4) || 3 + 3 < 5 / 3

Q3. The following declares a boolean variable that is true when z is smaller than both x, y, i.e., What's wrong with this:

```
int x = 42;
int y = 24;
int z = 10;
bool zSmallest = (z < x && y);
```

Q4. You're writing software for Dr Badadviz. After 20 years of research he found that a couple is compatible if the sum of their salaries (in pennies) when dividing by the sum of the weight (in lbs) is less than the number of fingers they have. Help Dr Badaviz write a program that prompts the user for the above, sets a boolean value `compatible` to true exactly when the couple is compatible.

Q5. In order to join the "Foo CEO Club", you need to have $1000000 in personal assets, is or has been a CEO of at least 3 companies, and the sum of your height in ft and weight in lbs must be at most 300, and finally either you have at least 5 houses or at least 2 jets.
Write a program that prompts the user for the above relevant information and sets a boolean variable to true exactly when the user is eligible for membership; print the value of this variable. Before you begin, here a test case.

Test case:

assets = 2000000
companies = 5
height = 6
weight = 200
houses = 4
jet = 0
result = NOT ELIGIBLE

NOW ... go ahead and write your program and test it against the test cases.