CS111

Introduction to Computing Science

Boolean Variables and Operators

Will we remember next time?
I wish I could put the way to go in my pocket!

Boolean Variables and Operators

- Sometimes you need to evaluate a logical condition in one part of a program and use it elsewhere.
- To store a condition that can be **true** or **false**, you use a Boolean variable.
- Sometimes you need to express a complicated condition.
- Boolean operations are used to express those conditions.

Leap years according to the Gregorian calendar (since 1582) are years that are exactly divisible by 400 or years that are exactly divisible by 4, except that years that are divisible by 100 are not leap years. Exactly divisible means that the remainder of the integer division is zero.

Boolean Variables and Operators

- Boolean variables are named after the mathematician George Boole.
- George Boole (1815–1864), was a pioneer in the study of logic.
- He invented an algebra based on only two values:

TRUE and FALSE .

Boolean Variables and Operators

Type **`bool`**

- In C++, the **`bool`** *data type* represents the Boolean type.
- Variables of type **`bool`** can hold exactly two values, denoted **`false`** and **`true`**.
- These values are **<u>not</u>** strings.
- There values are *definitely* **<u>not</u>** integers

They are special values, just for Boolean variables.

Boolean Variables

Example of defining a Boolean variable

```
bool isLegalAge = false;
```

- A Boolean variable named **`isLegalAge`**, initialized to **`false`**.
- It can be set by an intervening statement
- It can be used later to make a decision

Boolean Variables

Example of using a Boolean variable:

```
const int LEGAL_AGE = 21;
int student_age;
bool isLegalAge;

cout << “What’s your age? “;
cin >> student_age;

isLegalAge = (student_age >= LEGAL_AGE);
…
if (isLegalAge)
        cout << “Ok, you can drink”;
else
        cout << “No drinks for you”;
```

The right hand side is a condition.

It evaluates to “true” or “false”.

The result is then assigned to variable “isLegalAge”

Can be used as condition later on.

Boolean Variables

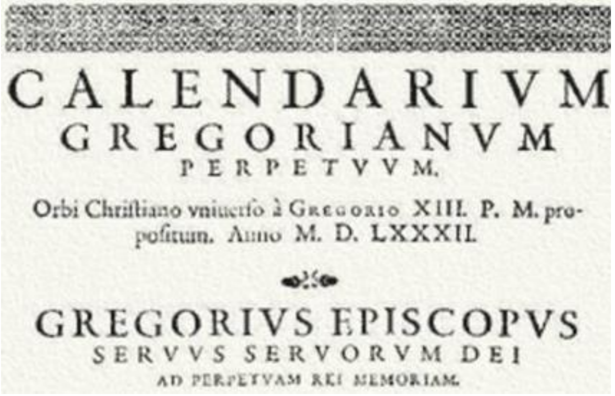
Sometimes bool variables are called “flag” variables.

The flag is either up or down.

C++ for Everyone by Cay Horstmann

Boolean Operators

- Suppose you need to write a program that decides if a year is a leap year.
- This not a simple test condition.
  - A year is a leap year if it is divisible by 400, or
    - Years that are divisible by 4,
    - But not by 100.



CALENDARIVM
GREGORIANVM
PERPETVVM.

Orbi Chriſtiano vniuerſo à GREGORIO XIII. P. M. propoſitum. Anno M. D. LXXXII.

GREGORIVS EPISCOPVS
SERVVS SERVORVM DEI
AD PERPETVAM REI MEMORIAM.

Boolean Operators

Complex Decisions

- When you make complex decisions, you often need to combine Boolean values.
- An operator that combines Boolean conditions is called a Boolean operator.
- Boolean operators take one or two Boolean values or expressions and combine them into a resultant Boolean value.

Boolean Operators

Complex Decisions

- Boolean Algebra allows you to compute with Boolean variables like Algebra with “normal variables”
- You have Boolean operations similar to addition and multiplication in “normal Algebra”.
- Some common Boolean operators are “AND”, “OR”, and “NOT”.

The Boolean Operator `&&` (and)

Boolean “AND”

- The Boolean expression

  a AND b

  evaluates to **true**, only if *a is true* **and** *b is true*.
- **Both have to be true.**
- In C++ we use the **`&&`** operator for “AND”.

The Boolean Operator `&&` (and)

Example

```
  if (shark_free && sunny)
  {
     cout << “Go swimming!";
  }
```

You can also combine conditions

```
if ((temperature >=0) && (temperature <=100))
  {
     cout << “Liquid Water";
  }
```

The Boolean Operator `||` (or)

Boolean “or”

- The Boolean expression

a or b

evaluates to **true**, if *a is true* **or** *b is true*.

- **At least one is true.**
- **Both could be true.**
- In C++ we use the `||` operator for “OR”.

“Either Or” means exactly one is true.

Do not confuse “OR” with “EITHER OR”!

The Boolean Operator || (or)

Example

```
if (sharks || rainy)
{
   cout << “Don’t swim!";
}
```

Don’t swim if there are sharks or if it is rainy. Or both.

You can also combine conditions

```
if ((temperature <0) || (temperature > 100))
  {
     cout << “No Liquid Water";
  }
```

The Boolean Operator || (or)

Do you want coffee or tea?
Yes!
Logic rules!
Nerd!
You can have too much logic with your coffee (or tea)

The Boolean Operatorion

Boolean “NOT”

- The Boolean expression

  NOT a

  evaluates to **true**, only if *a is false*.
- In C++ we use the **`!`** operator for “NOT”.

`if it is` ***`not`*** ***`shark free`*** **`don’t swim.`**

The Boolean Operatorion

Example

```
if (!shark_free)
  {
      cout << “Don’t swim!";
  }
```

if it is *not shark free* don’t swim.

```
if (!(temperature >=0))
  {
     cout << “Freezing.";
  }
```

This is the same as temperature<0

The Boolean Operators

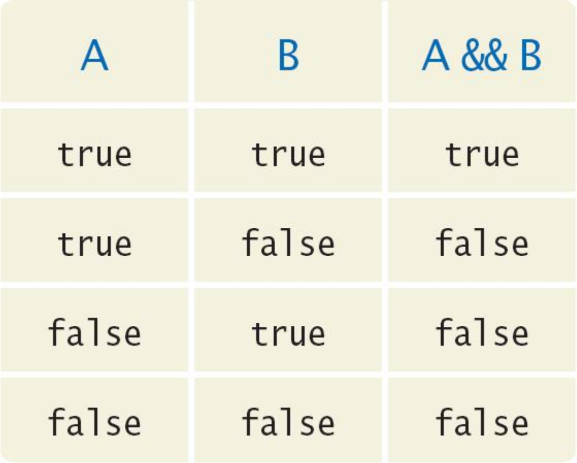
Combining operators

- You can combine different operators.
- Use parentheses to define the order of evaluation.

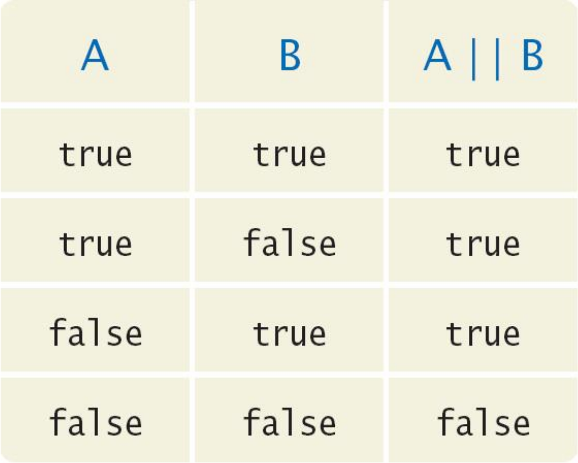
```
if ( !sharks && (warm || sunny))
{
   cout << “Go swimming!";
}
```

Boolean Operators

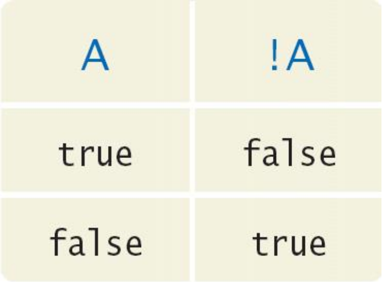
This information is traditionally collected into a table called a *truth table*:



A	B	A && B
true	true	true
true	false	false
false	true	false
false	false	false



A	B	A || B
true	true	true
true	false	true
false	true	true
false	false	false



A	!A
true	false
false	true

where A and B denote **`bool`** variables or Boolean expressions.

Boolean Operators – Some Examples

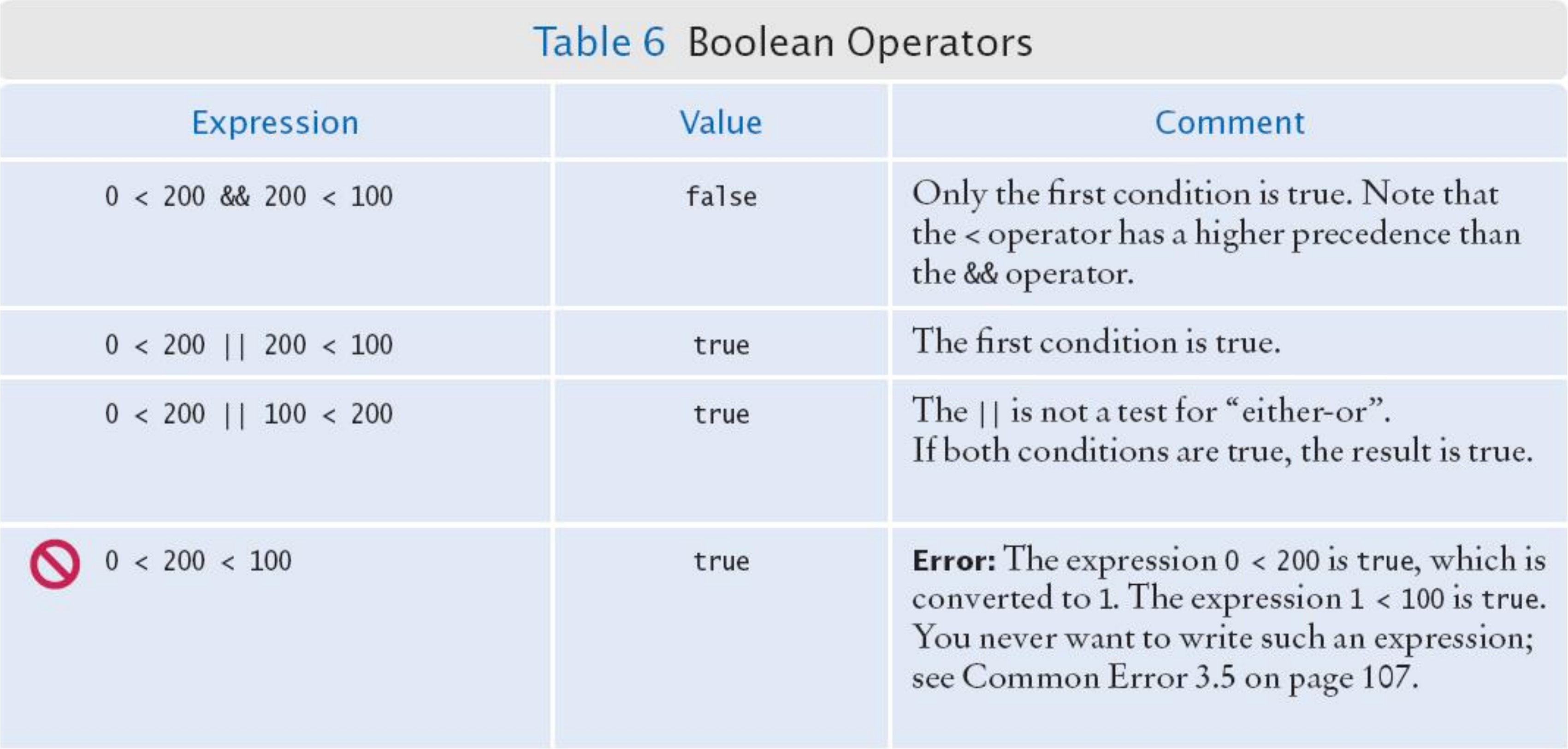
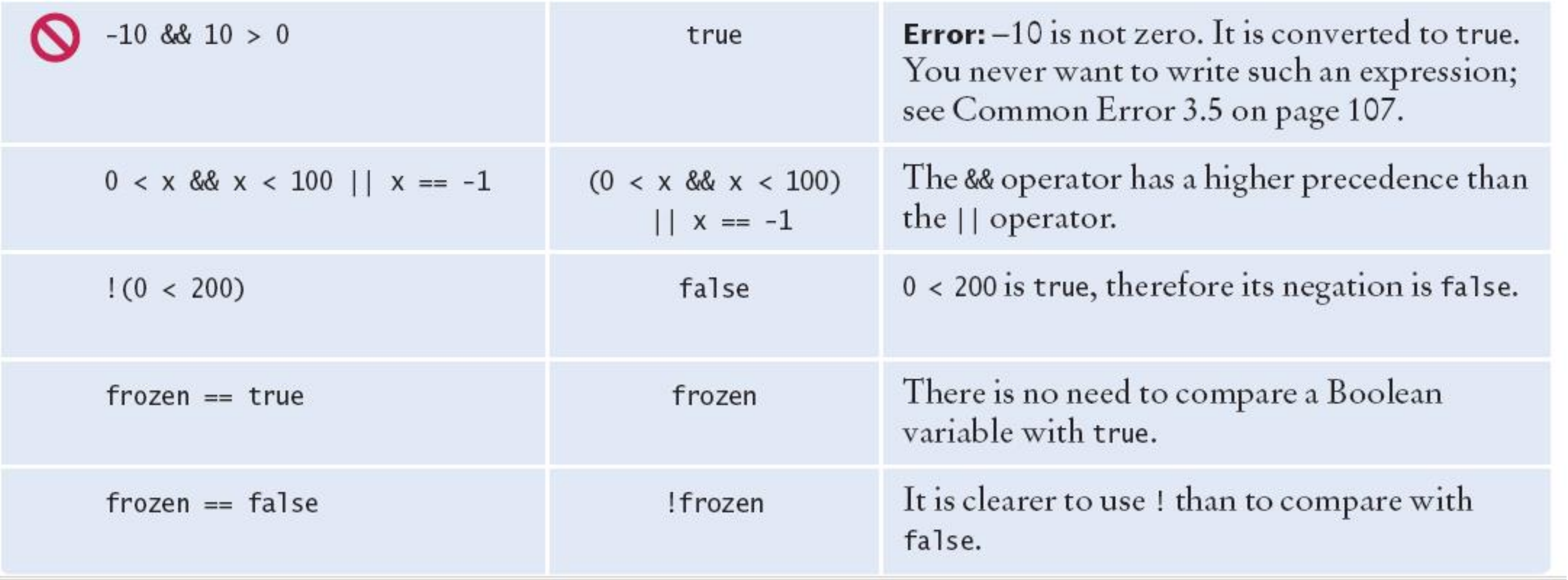


Table 6 Boolean Operators

Expression	Value	Comment
`0 < 200 && 200 < 100`	`false`	Only the first condition is true. Note that the < operator has a higher precedence than the && operator.
`0 < 200 || 200 < 100`	`true`	The first condition is true.
`0 < 200 || 100 < 200`	`true`	The || is not a test for "either-or". If both conditions are true, the result is true.
🚫 `0 < 200 < 100`	`true`	**Error:** The expression `0 < 200` is `true`, which is converted to 1. The expression `1 < 100` is `true`. You never want to write such an expression; see Common Error 3.5 on page 107.

Boolean Operators – Some Examples



🚫 `-10 && 10 > 0`	`true`	**Error:** –10 is not zero. It is converted to true. You never want to write such an expression; see Common Error 3.5 on page 107.

`0 < x && x < 100 || x == -1`	`(0 < x && x < 100)`
	`|| x == -1`
		The `&&` operator has a higher precedence than the `||` operator.

`!(0 < 200)`	`false`	`0 < 200` is `true`, therefore its negation is `false`.
`frozen == true`	`frozen`	There is no need to compare a Boolean variable with `true`.
`frozen == false`	`!frozen`	It is clearer to use `!` than to compare with `false`.

Common Error – Combining Multiple Operators

Consider the expression

```
if (0 <= temp <= 100)…
```

This looks just like the mathematical test:

$$0 \le \text{temp} \le 100$$

Unfortunately, it is not.

It might compile, but it is (almost) certainly wrong

Use: `if (0 <= temp  && temp <= 100)`…

Common Error – Combining Multiple Operators

Another common error, along the same lines, is to write

```
if (x && y > 0) ... // Error
```

instead of

```
if (x > 0 && y > 0) ...
```

(x and y are ints)

C++ for Everyone by Cay Horstmann

Common Error – Combining Multiple Operators

Naturally, that computation makes no sense.

(But it was a good attempt at translating:
“both x and y must be greater than 0” into
a C++ expression!).

Again, the compiler would not issue an error message.
It would do something you probably do not expect.

Common Error – Confusing `&&` and `||`

It is quite common that the individual conditions are nicely set apart in a bulleted list, but with little indication of how they should be combined.

Example

Concession fees apply if

- You are a student
- Younger than 12 years old
- A pensioner
- A staff member

`Is this an && or an || ?`

`Student and pensioner?`

Common Error – Confusing `&&` and `||`

It is quite common that the individual conditions are nicely set apart in a bulleted list, but with little indication of how they should be combined.

Example

You can apply for the grant if you

- are a staff member
- are at least 3 years employed
- have at least 3 recommendations
- have not been awarded before.

```
Is this
an && or
an || ?
```

```
Staff or 3
year
employed?
```

DeMorgan’s Law

Suppose we want to charge a higher shipping rate
if we don’t ship within the main islands of Fiji.

```
shipping_charge = 10.00;
if (!(country == "FIJI"
            && division != "Eastern" && division != "Rotuma"))
   shipping_charge = 20.00;
```

This test is a little bit complicated.

DeMorgan’s Law to the rescue!

DeMorgan’s Law

DeMorgan’s Law:

!(A && B) is the same as !A || !B

(change the && to || and negate all the terms)

!(A || B) is the same as !A && !B

(change the || to && and negate all the terms)

DeMorgan’s Law

DeMorgan’s Law allows us to rewrite complicated not/and/or messes so that they are more clearly read.

```
shipping_charge = 10.00;
if (country != "FIJI"
    || division == "Eastern" || division == "Rotuma"))
        shipping_charge = 20.00;
```

Ah, much nicer.

Input Validation with `if` Statements

You, the C++ programmer, doing Quality Assurance

(by hand!)

Input Validation with `if` Statements

Input validation is an important part of working with live human beings.

It has been found to be true that, unfortunately, all human beings can mistke makez.

Input Validation with `if` Statements

Let’s return to the elevator program and consider input validation.

Input Validation with `if` Statements

- Assume that the elevator panel has buttons labeled 1 through 20 (*but not 13!*).
- The following are illegal inputs:
  - The number 13
  - Zero or a negative number
  - A number larger than 20
  - A value that is not a sequence of digits, such as five
- In each of these cases, we will want to give an error message and exit the program.

Input Validation with `if` Statements

It is simple to guard against an input of 13:

```
if (floor == 13)
{
   cout << "Error: "
      << " There is no thirteenth floor."
      << endl;
   return 1;
}
```

Input Validation with `if` Statements

The statement:

```
return 1;
```

immediately exits the **`main`** function and therefore terminates the program.

It is a convention to return with the value 0 if the program completes normally, and with a non-zero value when an error is encountered.

Input Validation with `if` Statements

To ensure that the user doesn’t enter a number outside the valid range:

```
if (floor <= 0 || floor > 20)
{
   cout << "Error: "
      << " The floor must be between 1 and 20."
      << endl;
   return 1;
}
```

Input Validation with `if` Statements

Dealing with input that is not a valid integer is a more difficult problem.

What if the user does not type a number in response to the prompt?

‘F’ ‘o’ ‘u’ ‘r’ is not an integer response.

Input Validation with `if` Statements

When

```
cin >> floor;
```

is executed, and the user types in a bad input, the integer variable **`floor`** is not set.

Instead, the input stream **`cin`** is set to a failed state.

Input Validation with `if` Statements

You can call the **`cin.fail`** function to test for that failed state.

So you can test for bad user input this way:

```
if (cin.fail())
{
   cout << "Error: Not an integer." << endl;
   return 1;
}
```

Input Validation with `if` Statements

Later you will learn more robust ways to deal with bad input, but for now just exiting main with an error report is enough.

Here’s the whole program with validity testing:

Input Validation with `if` Statements

```
int main()
{
   int floor;
   cout << "Floor: ";
   cin >> floor;

   // The following statements check various input errors
   if (cin.fail())
   {
      cout << "Error: Not an integer." << endl;
      return 1;
   }
   if (floor == 13)
   {
      cout << "Error: There is no thirteenth floor." << endl;
      return 1;
   }
   if (floor <= 0 || floor > 20)
   {
      cout << "Error: The floor must be between 1 and 20." << endl;
      return 1;
   }
...
```

Input Validation with `if` Statements

```
   // Now we know that the input is valid
   int actual_floor;
   if (floor > 13)
   {
      actual_floor = floor - 1;
   }
   else
   {
      actual_floor = floor;
   }

   cout << "The elevator will travel to the actual floor "
      << actual_floor << endl;

   return 0;
}
```

Exercise: Leap Year

Write a program to check is a year is a leap year:

Leap years according to the Gregorian calendar (since 1582) are years that are exactly divisible by 400 or years that are exactly divisible by 4, except that years that are divisible by 100 are not leap years. Exactly divisible means that the remainder of the integer division is zero.

What is the pseudo code?

Exercise: Leap Year

The pseudo code is

```
If ((year divisible by 400) OR
    (year divisible by 4 AND
     year not divisible by 100))
   leap_year = true
Else
   leap_year = false
```

Exercise: Leap Year

What type do you take for **`year`** and **`leap_year`**?

Exercise: Leap Year

How do you express

```
((year divisible by 400)
OR
(year divisible by 4 AND year not divisible by 4))
```

Exercise: Leap Year

Finally add it to the if-condition with the assignments.

Summary

Use the `if` statement to implement a decision.

- The `if` statement allows a program to carry out different actions depending on the nature of the data to be processed.

Implement comparisons of numbers and objects.

- Relational operators (`< <= > >= == !=`) are used to compare numbers and strings.
- Lexicographic order is used to compare strings

Summary

Implement complex decisions that require multiple if statements.

- Multiple alternatives are required for decisions that have more than two cases.
- When using multiple **`if`** statements, pay attention to the order of the conditions.

Summary

Implement decisions whose branches require further decisions.

- When a decision statement is contained inside the branch of another decision statement, the statements are *nested*.
- Nested decisions are required for problems that have two levels of decision making.

Chapter Summary

Design test cases for your programs.

- Each branch of your program should be tested.
- It is a good idea to design test cases before implementing a program.

Use the `bool` data type to store and combine conditions

- The **`bool`** type bool has two values, **`false`** and **`true`**.
- C++ has two Boolean operators that combine conditions:

  `&&` (*and*) and **`||`** (*or*).
- To invert a condition, use the **`!`** (*not*) operator.
- De Morgan's law tells you how to negate **`&&`** and **`||`** conditions.

Chapter Summary

Apply `if` statements to detect whether user input is valid.

- When reading a value, check that it is within the required range.
- Use the fail function to test whether the input stream has failed.