Topic 7

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
- Variables of type **`bool`** can hold exactly two values, **`false`** or **`true`**.
  - **<u>not</u>** strings.
  - **<u>not</u>** integers; they are special values, just for Boolean variables.
  - BUT actually **`false`** is zero, and any non-zero value is treated as **`true`**.

Boolean Variables

Here is a definition of a Boolean variable, initialized to `false`:

```
bool failed = false;
```

It can be set by an intervening statement so that you can use the value *later* in your program to make a decision:

```
// Only executed if failed has
// been set to true
if (failed)
{
   ...
}
```

Boolean Operators Motivation

- Suppose you need to write a program that processes temperature values, and you want to test whether a given temperature corresponds to liquid water.
  - At sea level, water freezes at 0 degrees Celsius and boils at 100 degrees.
- Water is liquid if the temperature is greater than zero and less than 100.
- This not a simple test condition.

Boolean Operators

- 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.

The Boolean Operator `&&` (and)

- In C++, the **`&&`** operator (called *and*) yields **`true`** only when *both* conditions are **`true`**.

```
if (temp > 0 && temp < 100)
{
   cout << "Liquid";
}
```

- If **`temp`** is within the range, then both the left-hand side *and* the right-hand side are **`true`**, making the whole expression's value **`true`**.
- In all other cases, the whole expression's value is **`false`**.

The Boolean Operator || (or)

- The `||` operator (called *or*) yields the result **`true`** if at least one of the conditions is **`true`**.
  - This is written as two adjacent vertical bar symbols.

```
if (temp <= 0 || temp >= 100)
{
    cout << "Not liquid";
}
```

- If *either* of the expressions is **`true`**, the whole expression is **`true`**.
- The only way “Not liquid” won’t appear is if *both* of the expressions are **`false`**.

The Boolean Operator `!` (not)

- Sometimes you need to invert a condition with the logical *not* operator.
- The `!` operator takes a single condition and evaluates to **`true`** if that condition is **`false`** and to **`false`** if the condition is **`true`**.

```
if (!frozen) { cout << "Not frozen"; }
```

- "Not frozen" will be written only when frozen contains the value **`false`**.
- **`!false`** is **`true`**.

Boolean Operator Truth Tables

- This information is traditionally collected into a table called a *truth table*, where A and B denote **`bool`** variables or Boolean expressions.

`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`

Boolean Operator Examples: Table 6 (Part 1)

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.

Boolean Operator Examples: Table 6 (Part 2)

Expression	Value	Comment
`-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.
`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 Relational Operators

- Consider the expression

  `if(0 <= temp <= 100)`...

  This looks just like the mathematical test:

  $0 \leq temp \leq 100$

- Unfortunately, it is not. It will compile OK, but will not run the way you expect.

- DO NOT USE THAT SYNTAX IN C++. INSTEAD, USE the Boolean && operator to combine two pair compares:

  `if(0 <= temp && temp <= 100)`...

Combining Multiple Relational 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 **`int`**s)

Confusing && and || Conditions

- 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.
- Our tax code is a good example of this.

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

- Consider these instructions for filing a tax return.
- You are of single filing status if any one of the following is true:

  You were never married.

  You were legally separated or divorced on the last day of the tax year.

  You were widowed, and did not remarry.

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

- Since the test passes if any one of the conditions is **`true`**, you must combine the conditions with the **`or`** operator.

Confusing `&&` and `||` Conditions, continued

- Elsewhere, the same instructions:
- You may use the status of married filing jointly if all five of the following conditions are true:

  Your spouse died less than two years ago and you did not remarry.

  You have a child whom you can claim as dependent.

  That child lived in your home for all of the tax year.

  You paid over half the cost of keeping up your home for this child.

  You filed a joint return with your spouse the year he or she died.
- `&&` or an `||`?
- Because all of the conditions must be **`true`** for the test to pass, you must combine them with an **`&&`**.

Short Circuit Evaluation

When does an expression become **`true`** or **`false`**?

`expression && expression && expression &&` …

With &&’s, we can stop after finding the first **`false`**.

`expression || expression || expression ||` …

With **`||`**’s, we can stop after finding the first **`true`**.

This is called *short circuit evaluation*

DeMorgan’s Law

- Suppose we want to charge a higher shipping rate if we don’t ship within the continental United States.

```
shipping_charge = 10.00;
if (!(country == "USA"
      && state != "AK"
      && state != "HI"))
   shipping_charge = 20.00;
```

This test is a little bit complicated.

DeMorgan’s Law to the rescue!

DeMorgan’s Law, continued

- 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 != "USA"
     || state == "AK"
     || state == "HI")
   shipping_charge = 20.00;
```

Ah, much nicer.

But how did they do that?

DeMorgan's Law Equivalencies

- 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)

Simplification Examples: DeMorgan's, et al

- Simplify the following logical conditions:

```
int n; bool b; //definition of variables


n < 5 || n == 5

n <= 5 && n != 5

n <= 5 && n >= 5

n <= 5 || n >= 5

!(n <= 5)

!!b

b == true

b == false
```