CS 110 SI – Winter 2016

Session 3: Booleans, Conditions and Switches
Session Outline

1. Quick Review
   1.1 Type casting
   1.2 If / then / else
   1.3 Boolean Logic
   1.4 Switch case

2. Examples

3. TIWIKWITCS110 – Magic Numbers
Type Conversion

- There are two ways to convert, explicit and implicit

- Implicit: Let the compiler choose for me how to convert data (not good)

- Explicit: Let me choose how to convert data (better)
Implicit conversion (aka coercion)

int x = 3.76
float f = 3
char c = 95
Explicit conversion (aka casting)

• C style:
  (int) 3.76

• Function style:
  int (3.76)

• C++ way:
  static_cast <int> (3.76)
Trace this code block...

```c++
int x = 4;
int y = 7.5;
int z = y / x;
cout << z;
```

• Is this an error? If so what kind?
Trace this code...

```cpp
int x = 3;
float y = 7.5;
int z = x * y;
cout << z;
```

- What type of error is this?
// find the midpoint of two numbers
int hi = 21;
int lo = 6;
float mid = ((hi - lo) / 2) + lo;
cout << mid;

• What type of error is this?
• Can we fix it with casting?
Trace this code...

// find the midpoint of two numbers
int hi = 21;
int lo = 6;
float mid = ( float (hi - lo) / 2 ) + lo;
cout << mid;

*Fixed with the magic of casting*
Booleans

Represents logic

bool newFlag = false;

A single bit in your memory
Symbolically either true or false
Comparisons

- You can make booleans with comparisons
- The following expressions evaluate to booleans:
  
  \[
  > \quad \geq \\
  < \quad \leq \\
  == \quad !=
  \]
- As with math, comparators are defined by their data types.
Test Equality

- Denoted by the double equal signs
  
  `==`

- Defined by the data types.

Think about the following:

'\text{A}' == 'a'

“\text{A string}” == “\text{a string}”

\[ 3.0000000000000009 == 3 \]
Comparison's are based data type

- For the most predictable output, only run comparisons when the data type is the same.

- If you run comparisons on mismatched types, expect unpredictable output.

- Comparisons that make sense for int, float:
  
  $\geq, >, \leq, <, ==, !=$
Boolean Example

int x = 6;
int y = 7;
bool isXGreaterThanY = x > y;
Boolean Algebra

- Booleans can also be combined for more complex comparisons.
  - **AND** `&&`  Return false if any value is false, else return true
  - **OR** `||`  Return true if any value is true, else return false
  - **NOT** `!`  Return false if true, else return true.
## Truth Tables - &&

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A &amp;&amp; B</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
</tbody>
</table>
### Truth Tables - ||

<p>| A   | B    | A || B |
|-----|------|------|
| true| true | true |
| true| false| true |
| false| true | true |
| false| false| false |</p>
<table>
<thead>
<tr>
<th>A</th>
<th>!A</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
</tr>
</tbody>
</table>
C++ will only evaluate code until it can tell for sure if the outcome will be true or false. Try evaluating the following:

- bool amIADog = (legs == 4 && hasFur == true && eatsDogFood == true);

- How many tests do you need to do for a dog?
- How many tests do you need for a human?
How many comparisons?

• How many comparisons to evaluate the following:

  bool ILikeCS = (ICodeForFun == true || ITeachCS110SI == true || ICanReadBinary == true)
Strange Logic

• How would computers reason out the following:
  - ( 12 > 5 ) || ( The sky is purple on Venus )
  - ( 11 < 6 ) && ( I got less than 3 hours of sleep last night)
  - The earth is round || The earth is flat
  - The earth is flat || The earth is round
  - The earth is flat && The earth is round
Write a comparison to test...

• Any of x, y, or z are greater than 10.

• If a printer is ready to print, and there are more than zero jobs waiting for it.

• If all of x, y, or z are between 0 and 100.

• If a and b are true, but c is false.
Branching Your Code

- Two main mechanisms to branch in C++
  - if / else / elseif
  - switch / case
if (boolean or condition) {
    // this runs if the first condition is true
} else if (boolean or condition) {
    // this runs if the second condition is true
} else {
    // this runs if the first and second condition are false
}
Example

```cpp
int x;
cin >> x;
if ( x > 0 ) {
    cout << "the number is positive";
} else if ( x < 0 ) {
    cout << "the number is negative";
} else {
    cout << "the number is zero";
}
```
Nesting if...s...

if ( money > 0) {
    if (inJail == true) {
        cout << “Pay your bail or roll some doubles!”;
    } else {
        cout << “Your turn to roll”;
    }
} else {
    cout << “You're bankrupt!”;
}
Switch Case

switch (x) {
    case 1:
        // do things for 1 here
        break;
    case 2:
        // do things for 2 here
        break;
    default:

}
Couple o' things...

- Switch / Case only works with integer-like datatypes (int, char)
- The break keyword prevents subsequent cases from being executed. C++ will keep executing code until it hits a break.
  - Break is optional on the last case.
- The optional default label will get called if none of the other options evaluate to true.

The case checks must be hard coded literals.
Lazy cases

```cpp
switch (x) {
    case 1: // fall through
    case 2: // keep falling through
    case 3: // still falling through
        cout << "the number is 1, 2, or 3...";
        break; // caught!
    default:
        cout << "the number is not 1, 2 or 3";
}
```
PIN Example:

- Write a program which allows the user to enter 4 single digit numbers.
- Check if all 4 digits match stored constants.
  - If so, let the user in.
  - If not, do not let the user in.
• Magic Numbers is a code-word for literals scattered throughout your program. Take a look at the following code:

```java
• if ( m == 1 && d < 31 ) {
    // do things here
} else if (m == 9 && d < 30) {
    // do more things here
}
```
Here's the same code “translated” using constants:

```java
if (month == JAN && days < DAYS_IN_JAN) {
    // do things here
} else if (month == SEPT && days < DAYS_IN_SEPT) {
    // do more things here
}
```
Pros of using Constants

• Don't need to remember cumbersome numbers

• Need to comment your code a lot less

• Makes it easy for others to decipher

• Makes it easy for you to decipher later on
Pros of using Constants

• Reduce the dreaded “off by one” error

• Makes your code more reusable

• Can help reduce type errors.