COMP 1633: Intro to CS II

Booleans and Decisions

Charlotte Curtis

January 24, 2024

Where we left off

- Pass by reference
- Testing with functions
- Intro to decisions

```
if (boolean_expression) {
    // code to execute if true
} else {
    // code to execute if false
}
```

Today's topics

- Boolean expressions
- `if-else` statements
- Some C++ specific boolean behaviour
- Intro to loops

> *Textbook Sections 2.4, 3.1-3.2*

The `bool` data type

- A primitive just like `int` and `double`
- `bool` can be declared and initialized like any other primitive

```
bool thunder_only_happens_when_its_raining = true;
```

- `bool` can also be returned from a function

```
bool is_valid_account_number(int account_number);
```

- But often expressions are used directly without assigning to a variable

```
if (temperature < 0) {
    cout << "It's freezing!\n";
}
```

Reading and printing booleans

- `bool` values can't really be read in or printed out, but they can be **implicitly converted** to `int` values
- Caution: there's no loss of precision, so no compiler warning!
    - `false` is converted to `0`, `true` is converted to `1`
    - `0` is converted to `false`, and any other number is `true`
- Safer to read a `char` and convert to `bool` explicitly

```
char c;
cin >> c;
bool is_valid = c == 'y' || c == 'Y';
```

Boolean operators

Python	C++	Description
`and`	`&&`	Logical and
`or`	`||`	Logical or
`not`	`!`	Logical not

- Same behaviour and precedence as Python, just different symbols
- Like Python, **short circuit** evaluation is used

- Example: assign a boolean `timed_out` that is a function of two `ints`: `total_time` and `num_records`.
- `timed_out` should be `true` if the **time per record** exceeds 1 second, and `false` otherwise.

`cout` and precedence

- In Python, `print` is a function, so the whole expression is evaluated first:

```
print(x > 0 and x < 10) # prints True or False
```

- In C++, `<<` is an **operator**:

```
cout << x > 0 && x < 10; // what happens?
```

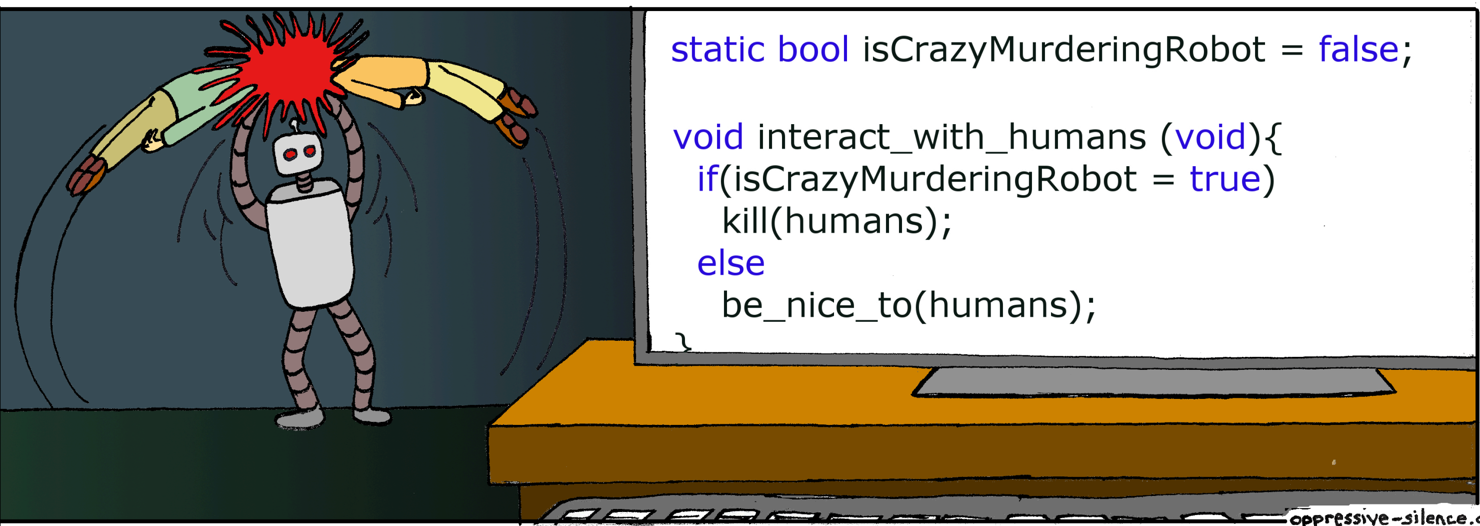
- Easiest solution: use parentheses, or assign to a variable:

```
cout << (x > 0 && x < 10);
```

`if` syntax

```
if (boolean_expression) {
    // code to execute if true
} else {
    // code to execute if false
}
```

- Remember the `{}` defines blacks in C++
- `boolean_expression` can be a compound condition, a function returning a `bool`, a single boolean variable... anything that evaluates to a `bool`
- Caution: sometimes things evaluate to a `bool` when you didn't expect it!



Caution!

- `=` is assignment, `==` is comparison
- Unlike Python, this is not a syntax error
- Better to avoid comparison with `bool`

Single line `if` statements

- If the code to execute is a single line, you *can* omit the curly braces

```
if (x > 0)
    cout << "x is positive\n";
```

- Ditto for `else`:

```
if (x > 0)
    cout << "x is positive\n";
else
    cout << "x is negative\n";
```

- This can be risky though - remember C++ doesn't care about indentation!

Nested `if` statements

Just like Python, you can nest `if` statements inside each other:

```
if (is_valid_account_number(account_number) {
    if (max_disk_usage > allotment) {
        // surcharge calculation
    }
}
```

- Indentation is not required, but it's a good idea
- Emacs will indent for you, but if it's not, that could mean you have an error

Multiple branching with `else if`

Python

```python
if x > 0:
    print("x is positive")
elif x < 0:
    print("x is negative")
else:
    print("x is zero")
```

C++

```cpp
if (x > 0) {
    cout << "x is positive\n";
} else if (x < 0) {
    cout << "x is negative\n";
} else {
    cout << "x is zero\n";
}
```

- No special `elif` keyword, just `else` followed by `if`
- As many `else if` branches as you like, including zero

Tricky mistakes

- `;` after `if` statement

```
if (x > 0);
    cout << "x is positive\n";
```

- The "dangling `else`" problem

```
if (x > 0)
    if (y > 0)
        cout << "x and y are positive\n";
else
    cout << "x is negative\n";
```

`if` statement check-in

In the following code snippet, `x` has a value of 15. What is the output?

A. `Fizz`
B. `Buzz`
C. `FizzBuzz`
D. Nothing
E. Error

```
if (x % 3 == 0)
    cout << "Fizz";
if (x % 5 == 0)
    cout << "Buzz";
```

A trickier one

What is the output of the following code snippet? `x` is again 15.

A. `x is 0`

B.
```
x is 0
Try again
```

C. `Try again`

D. Nothing

E. Error

```
if (x == 0)
    cout << "x is 0\n";
    cout << "Try again\n";
```

Branching in functions

C++ does not restrict you to a single `return` statement in a function:

```
double relu(double x) {
    if (x > 0)
        return x;
    else
        return 0;
}
```

- Multiple returns can make code harder to read and debug, though unlike Python, the compiler will protect you from a forgotten `return`
- My recommendation: stick to a **single** **`return`**, unless it's a "guard clause"
    - return at the **start** or the **end** of the function, not in the middle

Tangent: Guard clauses

A "guard clause" is an `if` statement that returns early if inputs are invalid

```
bool is_valid_account_number(int account_number) {
    if (account_number < 0) {
        return false;
    }

    // rest of function
}
```

- This can prevent nesting and make code easier to read
- Guard clauses should be short and at the **very start** of the function

And now, loops!

```python
while condition:
    # code to execute
```

```c
while (condition) {
    // code to execute
}
```

- You basically know `while` loop syntax already! Just remember:
- Each loop has at least one **loop control variable** (LCV)
- The LCV must be **initialized** prior to the loop
- The LCV must be **updated** inside the loop
- Eventually the **condition** must become `false` to exit the loop

Complete `while` loop example

```
int x = 1;                  // Initialization
while (x <= 100) {          // Condition
    if (x % 3 == 0)
        cout << "Fizz";
    if (x % 5 == 0)
        cout << "Buzz";
    cout << "\n";
    x++;                    // Update
}
```

- Forgetting to update the LCV leads to an **infinite loop**
- Initializing with the wrong value can lead to the loop never executing

Not a complete FizzBuzz implementation - what is this missing?

`for` loops - a bit more different

```python
for i in range(10):
    # code to execute
```

```java
for (int i = 0; i < 10; i++) {
    // code to execute
}
```

- Notice the semicolons! Inside the parentheses, there are three **statements**:
  i. Initialization
  ii. Condition
  iii. Update
- The LCV is declared inside the loop, and only exists inside the loop
- BUT this isn't actually mandatory - it's a good idea though

FizzBuzz as a `for` loop

Since FizzBuzz is counting from 1 to 100, it's a good candidate for a `for` loop:

```
for (int x = 1; x <= 100; x++) {
    if (x % 3 == 0)
        cout << "Fizz";
    if (x % 5 == 0)
        cout << "Buzz";
    cout << "\n";
}
```

- `for` loops help protect you from forgetting to initialize or update the LCV
- More readable for counted scenarios, as all 3 steps are in one place
- BUT you ~~can't~~ shouldn't use a `for` loop for **event controlled** repetition

Coming up next

- Lab: Decisions
- Lecture: Loops in more detail, plus new loop constructs and C++ gotchas
- Get cracking on assignment 1! 🎉

> *Textbook sections 3.3-3.4*