BITS & BOBS

ADAM SWEENEY
CS 211

INTRODUCTION

- A little review
- A few small new things
- A few tips

AGENDA

- Boolean expression review
  - Partial order of operations
- Scope
- Choosing & designing loops
- Debugging loops

BOOLEAN EXPRESSION REVIEW

QUESTION

- Evaluate the following: `!(false || true)`

ANSWER

- Evaluate the following: `!(false || true)`
  - `(false || true) -> true`
  - `!(true) -> false`
  - Final answer: false

PARTIAL ORDER OF OPERATIONS

The unary operators +, -, ++, --, !

The binary arithmetic operators *, /, %

The binary arithmetic operators +, -

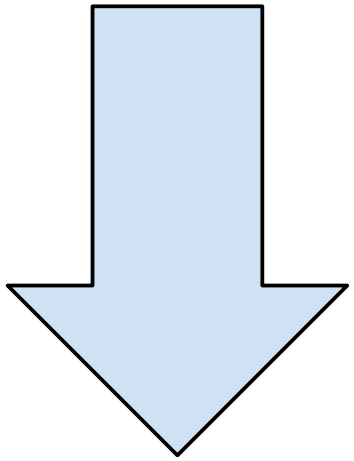
The Boolean operators <, >, <=, >=

The Boolean operators ==, !=

The Boolean operator &&

The Boolean operator ||

Highest Precedence
(done first)



Lowest Precedence
(done last)

CONSIDER THE FOLLOWING

- An if-else block that requires checking that a timer is under a certain limit
- Assume time = 36 and limit = 60
- How will it evaluate?

```
if (!time > limit) {
    // Do something
} else {
    // Do something else
}
```

UNTANGLING PRECEDENCE

- `!` is evaluated first
- `!36 -> false`
  - 36 evaluates to true, since it is not zero
- `false` is converted to 0 for the integer comparison
- `0 > 60 -> false`
- Final answer: `(!time > limit) -> false`
- How can we fix it?

SOME SOLUTIONS

- Enforce the correct order of operations
  - `(!time > limit)` -> `(!(time > limit))`
- Avoid `!` altogether
  - `(time <= limit)`
  - Generally easier to read and understand

SCOPE

BLOCKS

- Generally, everything between { and }, inclusive, is called a block
  - Functions are still called functions, this applies more to smaller pieces of code
  - Compound statements used in loops, if-else, etc.
- Many times, new variables are needed within blocks
- Knowing about scope will save a lot of headache

LOCAL SCOPE

```
1 #include <iostream>
2 int main() {
3     int count = 2;
4     for (int i = 1; i <= 3; i++) {
5         int count = 4 + i;
6         std::cout << count << ‘\n’;
7     }
8     std::cout << count << ‘\n’;
9 }
```

LOCAL SCOPE OUTPUT

```
1 #include <iostream>
2 int main() {
3     int count = 2;
4     for (int i = 1; i <= 3; i++) {
5         Int count = 4 + i;
6         std::cout << count << ‘\n’;
7     }
8     std::cout << count << ‘\n’;
9 }
```

- Output:

```
5
6
7
2
```

BIGGER BLOCK SCOPE

```
1 #include <iostream>
2 int main() {
3     int count = 2;
4     for (int i = 1; i <= 3; i++) {
5         count = 4 + i;
6         std::cout << count << ‘\n’;
7     }
8     std::cout << count << ‘\n’;
9 }
```

BIGGER BLOCK SCOPE OUTPUT

```
1 #include <iostream>
2 int main() {
3     int count = 2;
4     for (int i = 1; i <= 3; i++) {
5         count = 4 + i;
6         std::cout << count << ‘\n’;
7     }
8     std::cout << count << ‘\n’;
9 }
```

- Output:

```
5
6
7
7
```

CHOOSING & DESIGNING LOOPS

WHAT LOOP SHOULD I CHOOSE?

- Up to you
- Different loops feel more natural in certain scenarios
- Need to visit every element, or know how many iterations are needed?
  - A for loop
- Don't know when the loop will end?
  - A while or do-while loop
- Need to execute at least once?
  - A do-while loop

SOME QUICK TIPS

- Loop variables need to be initialized
- "Just avoid infinite loops"
- The `break` statement is not exclusive to `switch`
  - Can be used in any loop to prematurely exit
  - Do NOT rely on `break` for normal exits
  - `break` is the exception, not the rule

DESIGNING A LOOP

- Three areas of focus
  - The initialization
  - The body
  - The end conditions
- The easiest way:
  - Say what you want to do in plain language
  - Create pseudo-code
  - Translate pseudo-code to real code

AN EXAMPLE

- Get a number from the user that indicates how many data entries they intend to make, then return a sum of all the data
  - Repeat the problem in your own words
  - "I need to find out many data points there are, and then ask for data that many times and update the sum each time"
  - Repetition is inferred (loops!), along with a way to calculate the sum

OUR FIRST DRAFT

```
get count from user
repeat count times
    get number
    add number to running total
output total sum
```

- This first draft of pseudo-code describes the entire problem
- It gets us close enough to start writing some real code

TRANSLATE THE EASY STUFF

```
std::cin >> count;
repeat count times
    get number
    add number to running total
std::cout << sum << '\n';
```

- This code is not complete
- Getting it to compile is "busy" work
- We did the hard work already when we figured out the algorithm
- Focus on the loop now

CONSIDER THE LOOP

```
std::cin >> count;
repeat count times
    get number
    add number to running total
std::cout << sum << '\n';
```

- If we know how many times we need to loop, a for loop seems the most natural
- Remember, all loops are interchangeable, some just feel better in different situations

WORK SMART

```
std::cin >> count;
for (int i = 0; i < count; ++i)
{
    std::cin >> value;
    sum += value;
}
std::cout << sum << '\n';
```

- If we know how many times we need to loop, a for loop seems the most natural
- We need to consider what scope makes sense for variables value and sum
- The hard work was done before we wrote any code

FOUR WAYS TO END A LOOP

- By size
  - If you know how many iterations you need, you loop that many times
- Ask before iterating
- Sentinel value
  - A value that can never occur under normal circumstances
  - Not always possible
- Run out of input
  - Most common when reading files

DEBUGGING LOOPS

MISTAKES HAPPEN

- Off-by-one errors
  - Loop iterates one too few or one too many times
  - Check start values and end conditions
  - If a loop can iterate zero times, is it handled appropriately
- Infinite loops
  - Typically an error in Boolean expression or update action
  - Also setting your termination action as a test for equality
    - Especially bad with floating point types due to decimal approximation

DEBUGGING TIPS

- Ensure problem is in the loop
- Trace variables
  - Print to screen with pertinent information
- Always Be Testing
  - Don't wait for code to break
  - Don't "throw" code at problems to see if anything sticks
  - Testing is a controlled process with feedback to guide us