Operator Precedence
Boolean Expression Pitfalls
Debugging

Luke Sathrum – CSCI 20

Midterm #1

- September 28th
- 1 ½ hours
- Combination of
  - Multiple Choice
  - Fill in the blank
  - Short / Long Answer
  - Code
- Allowed to have a 3 x 5 handwritten notecard
  - Front and Back

Today's Class

- Precedence of Operators
- Short Circuit Evaluation
- Integer Values as Boolean Values
- Assignment vs. Equality (= vs. ==)
- Debugging
- Common Errors

Precedence of Operators

Precedence of Operators

- There is precedence with Boolean expressions
  1. `!`
  2. `<  ==  !=` (Relational Operators)
  3. && (AND)
  4. || (OR)
- There is precedence with all other operators as well

□ http://en.cppreference.com/w/cpp/language/operator_precedence

Notes on Precedence

- Assignment Operations are done right to left
  - `x = y = z` means `x = (y = z)`
- Binary Operations are done left to right
  - `x + y + z` means `(x + y) + z`
- Apply the KISS rule
- Example
  - `x + 1 > 2 || x + 1 < -3`
  - What is the order we evaluate this?

Notes on Precedence

```
x + 1 > 2   ||  x + 1 < -3
```

Summary

- All operators have an order of precedence
- Try to keep thing simple
  - Don’t do 50 operations all on one line
  - Use parenthesis

Sample Code

- Showing Precedence
  - `precedence.cpp`

Boolean Expression Pitfalls

Short Circuit Evaluation

- Idea that we don’t always need to evaluate both sides of an **&&** or **||** operation
- Why do we care?

Short Circuit Evaluation

```
if ((pieces / kids) >= 2)
  cout << "Each child may have two pieces";
```

- vs.

```
if ((kids != 0) && ((pieces / kids) >= 2))
  cout << "Each child may have two pieces";
```

Integer Values as Boolean Values

- Integer values can be converted to Boolean Values
- **`0`** is equivalent to **`false`**
- All other values are equivalent to **`true`**
  - We avoid doing this
- Sometimes a logical error will cause this problem

Integer Value Example

- We have a program that has 2 variables
  - **overall_time**
  - **limit**
- As long as time hasn't reached the limit we continue to run the program

```
if (!overall_time > limit){
  overall_time++;
}
```

Using = in place of ==

```
if(x = 12)
  //Do Something
else
  //Do something else
```

- What does this evaluate to?
- In C++ an assignment operation returns the value of the right side
- **`x = 12`** evaluates to **`12`**
- So this expression always evaluates to?

Summary

- C++ uses short-circuit evaluation
- Integers values can be used as Boolean values
  - We avoid doing this
- Don’t use assignment when you meant to use equality

Sample Code

- Boolean Expression Pitfalls
  - `pitfalls.cpp`

Debugging

Debugging

- A mistake in a program is called a bug
- The process of eliminating bugs is called debugging
- First bug was a moth
- There are 3 kinds
  - Syntax
  - Run-Time
  - Logic

Syntax Errors

- Result from violation of language’s syntax
  - Grammar rules of the language
- Things like
  - Forgetting a semicolon
  - Not closing your curly braces
  - Misspelling keywords

Syntax Errors

- Syntax errors are detectable by the compiler
- Will tell you where the error is
- Gives its best guess as to what the error is
  - Can be incorrect
- Never incorrect about a violation of syntax
- Good at determining close to where your error is

Syntax Errors

- Always correct the highest error first
  - Lowest line number
- Other errors may go away if you fix the first one
- These errors cause your program not to compile

Syntax Warnings

- Sometimes you’ll get a warning instead of an error
- Means you’ve done something that technically isn’t an error
  - Unusual enough to indicate a mistake
- Things like
  - Declaring a variable and not using it
  - Using an uninitialized variables
- Good to treat compiler warnings as errors

Run-Time Errors

- Detectable when the program is running
- Many run-time errors have to do with numeric calculations
- The most common is divide by zero
- May not crash your program every time

Logic Errors

- Have to do with the logic of your program
- When the program doesn’t do what we want it to do
- For example
  - Using a **+** when we wanted to do multiplication
  - Using **=** when we wanted to use **==**
- The program will compile
- Hardest to diagnose
  - No help from the computer

Summary

- We debug our programs to remove errors
- We have three types of errors
  - Syntax
  - Run-Time
  - Logic

Sample Code

- Debugging C++ Code
  - `debug.cpp`

Debugging – Best Practices

Assuming

- Don’t assume your program is correct
- Run your program several times
  - Use different data sets
  - Give different types of input
- Will give you more confidence that the program is correct

How to Fix

1. Localize the problem
    - Is it at the top?
    - Middle?
    - Bottom?
    - In a branching statement?
    - In a loop?
    - In a function?

How to Fix

2. Watch your variables change
   - Called tracing
   - Simple tracing is done by **`cout`** statements
     - Output the value of the variables in question
3. Fix the error
4. Re-Test
   - Re-test after every change
   - Never Assume
     - There could be other errors
     - Your “fix” may have introduced a different error

Bad Code

- If you have to debug some poorly designed code
  - Sometimes better to start over
- Will make the program
  - Easier to read
  - Have less hidden errors
- This is usually faster than trying to fix the bad code

Assertions

- Allow us to test our code by asserting if something is correct
- Basically a statement that is either true or false
  - The sky is blue today
  - The door is locked
  - The variable is not zero
- Use to document and check correctness of programs
- In C++ they are Boolean Expressions

Assertions

- Have a predefined Macro we use to check assertions
  - Macros are very similar to inline functions
- Part of the **`cassert`** library

```
#include <cassert>
```

- Syntax

```
assert(boolean_expression);
```

Assertions

- When the Macro is invoked it checks the Boolean Expression
- If true
  - Nothing happens
- If false
  - Default behavior is to end the program
  - Also outputs an error

Assertions

- You can turn assertions on/off
- Usually on for debugging
- Usually off for the end user
- Use
  - `#define NDEBUG`
- Put it before you include **`cassert`**
  - `#define NDEBUG`
  - `#include <cassert>`
- Just remove it to use your asserts again

Summary

- Don’t assume your code is working
- To fix our code
  1. Localize the problem
  2. Trace
  3. Fix
  4. Re-Test
- Don’t waste your time fixing bad code
- Assertions allow us to check for a certain condition

Sample Code

- Assertions
  - `assert.cpp`

Review

- Operators have precedence
  - KISS
- Short Circuit Evaluation says?
- Two Boolean Pitfalls
  - Integers vs. Booleans
  - Assignment vs. Equality
- Three types of coding errors
  - Syntax
  - Runtime
  - Logic
- __________ can help make it easier to debug your code