12. Compound Boolean Expressions

CPSC 120: Introduction to Programming
Pratishtha Soni~ CSU Fullerton

Agenda

0. Sign-in sheet
1. Technical Q&A
2. Logical Operators
3. Comma Operator
4. Extraction Validation (Simplified)
5. Number Ranges

1. Technical Q&A

Technical Q&A

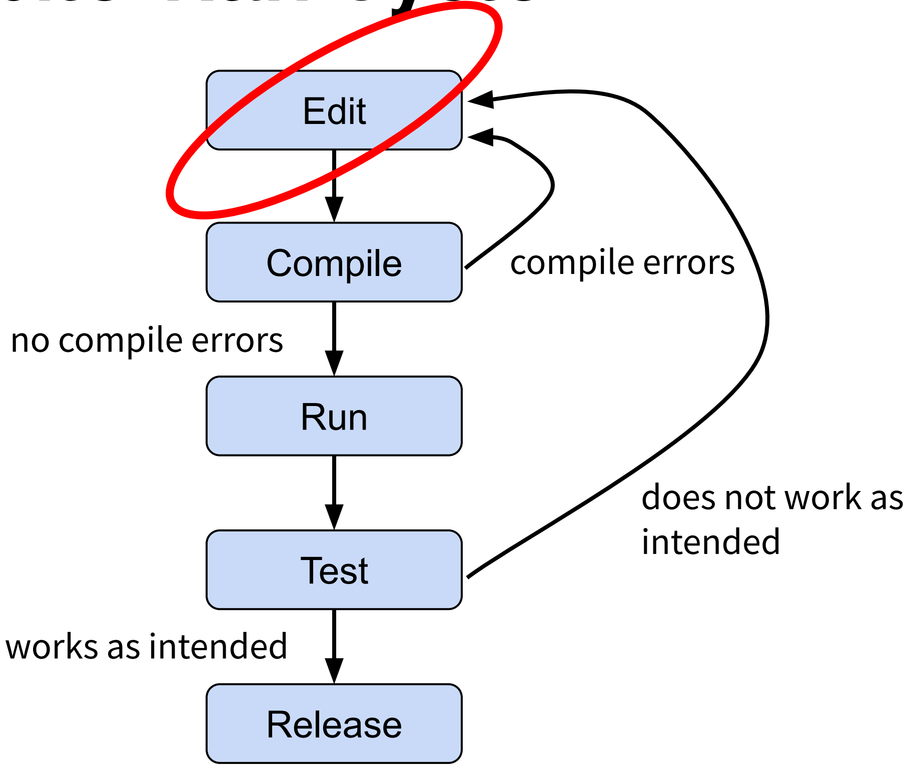
Let’s hear your noted questions about...

- This week’s Lab
- Linux
- Any other technical issues

Reminder: write these questions in your notebook during lab

2. Logical Operators

The Edit-Compile-Run Cycle



Logical Operators

- Recall: `bool` is data type for `true`/`false`
- So far, only operators on `bool` are `==, !=, =`
- Now: operators to combine `bool`s with AND, OR, NOT
- **Predicate:** expression that produces a `true`/`false` `bool`
- More complex predicates like “player 1 has more points than player 2 and player 1 has not already forfeited”
- These operators go in Boolean expressions e.g.
  if (*condition-expr* **)**

Summary of Logical Operators

<u>Operator</u>	<u>Name</u>	<u>Semantics</u>	<u>Example</u> (x and y are `bool` expressions)
&&	AND	**Both** operands are true	x && y
||	OR	Left operand **or** right operand **or** both	x || y
!	NOT	**Flip** true/false	! x

AND (&&)

- &&
  - true when both operands `true`
  - `false` otherwise
- Use for two **required** conditions
- Example: `memory` is at least 8 and `price` is at most 600:

```
memory >= 8 && price <= 600
```

OR (||)

- ||
  - `true` when one or both operands is `true`
  - `false` when both operands are `false`
- Use for two **alternative** conditions
- Example: player 1's score is greater than player 2's score, or player 2 is is ineligible:

```
player_1_score > player_2_score || player_2_eligible == false
```

NOT (!)

- `!`: unary operator; changes `true` to `false`; changes `false` to `true`
- Example: could rewrite

  ```
  player_1_score > player_2_score || player_2_eligible == false
  ```

  to

  ```
  player_1_score > player_2_score || !player_2_eligible
  ```

Example: Boolean Expressions in if Statements

```
int a{ 0 }, b{ 0 };
// ... input into a, b ...
if (a > 0 && b > 0) {
  std::cout << "both positive\n";
}
if (a < 0 || b < 0) {
  std::cout << "there is a negative\n";
}
if (!(a == b)) {
  std::cout << "different\n";
} else {
  std::cout << "same\n";
}
```

Precedence of Boolean Operators

- Without parenthesis, mixing AND, OR is confusing
- Q: in the predicate “soup or salad and coffee”, is it “(soup or salad) and coffee”, or “soup or (salad and coffee)”?
- && has higher precedence than ||
- `expr1 || expr2 && expr3`
  is equivalent to
  `expr1 || (expr2 && expr3)`
- `(x == 0 || x < 10 && y > 0)`
  is equivalent to
  `(x == 0 || (x < 10 && y > 0))`

Best Practice: Parentheses in Boolean Expr's

- **Best practice**: add parentheses around every part of a Boolean expression
- Don't need to memorize the && || precedence rule
- Instead of

  ```
  if (0 <= x && x <= 10)
  ```

  write

  ```
  if ((0 <= x) && (x <= 10))
  ```

- Instead of

  ```
  if (x == 0 || x < 10 && y > 0)
  ```

  write

  ```
  if (x == 0 || (x < 10 && y > 0))
  ```

Short-Circuit Evaluation

- Recall
  - `a && b`: both a and b are `true`
  - `a || b`: either a is `true`, or b is `true`, or both
- Sometimes computer can predict result from only a
  - `a && b`: if a is `false`, then `a && b` is automatically `false`
  - `a || b`: if a is `true`, then `a || b` is automatically `true`
- **Short circuit evaluation**: When evaluating `&&, ||`
  - Always evaluate left operand
  - Only evaluate right operand if necessary
- *"Short circuit"* = when right operand is skipped

Pitfall: Side Effects in Boolean Expressions

- Combining expressions with side-effects (e.g. ++), with short-circuit evaluation, can cause confusing bugs
- E.g:

```
if ((x > 0) && (++y > 0)) {
  cout << “both positive”;
}
```

- Looks like y is always pre-incremented by `++y`
- However that only happens when `(x > 0)`; if x ≤ 0, the && is automatically `false`, so `(++y > 0)` is not evaluated
- **Best practice**: do not use operators with side effects (++, --, *=, etc.) inside Boolean expressions

Pitfall: Bitwise AND/OR

- C++ has "bitwise" operators with similar names to `&&`, `||`
- `&` is bitwise AND (one `&` instead of `&&`)
- `|` is bitwise OR (one `|` instead of `||`)
- Bitwise operators are topics for MATH 170A, CPSC 240
- They do something different from `&&`, `||`
- For now, be careful to use the two-symbol operators `&&`, `||` not the one-symbol operators `&`, `|`
- E.g.
  ```
  if ((0 <= x) & (x <= 10)) // logic error
  ```
  should be
  ```
  if ((0 <= x) && (x <= 10))
  ```

3. Comma Operator

Comma Operator -- Never Use

expression:

left , *right*

Semantics:

1. Evaluate *left* and discard the result
2. Evaluate *right* and produce that value

Issues

- Confusing
- Almost entirely pointless

Example:

```
int a{ 5 }, b{ 1 }, c{ 0 };
c = a + 1, b + 1;  // discards a + 1
std::cout << c << “\n”; // prints 2
```

Pitfall: Comma Operator in Boolean Expr.

- In English, we use comma to mean AND e.g. “if x, y are both positive”
- C++ **does not** work this way
- Avoid temptation to put comma in Boolean Expressions
- Example in

  `if (x, y == 0)`

  `x, y` is a comma operator, so has the same value as just y

  so is equivalent to

  `if (y == 0)`
- Best practice: **never use comma operator**

Why does the comma operator exist?

- Misguided attempt to make increment statements more concise

  ```
  ++i, ++j;
  ```

- Confusing; readability more important
- Style guide says to just write two separate statements

  ```
  ++i;
  ++j;
  ```

- **Never use the comma operator**

4. Extraction Validation (Simplified)

Pattern: Validating Extraction

statements:

```
std::cin >> variable ;
if ( static_cast<bool>(std::cin) == false ) {
  std::cout << "command-error";
  return 1;
}
```

Example:

```
double price{0.0};
std::cout << "Enter price: ";
std::cin >> price;
if (static_cast<bool>(std::cin) == false) {
  std::cout << "error: unrecognized input\n";
  return 1;
}
```

Input/Output:

```
Enter price: nothing
error: unrecognized input
```

Review: NOT (!)

- `!`: unary operator; changes `true` to `false`; changes `false` to `true`
- Example: could rewrite

  `player_1_score > player_2_score || player_2_eligible == false`

  to

  `player_1_score > player_2_score || !player_2_eligible`

Pattern: Validating Extraction (Simplified)

statements:

```
std::cin >> variable ;
if ( !std::cin ) {
  std::cout << "command-error";
  return 1;
}
```

Example:

```
double price{0.0};
std::cout << "Enter price: ";
std::cin >> price;
if (!std::cin) {
  std::cout << "error: unrecognized input\n";
  return 1;
}
```

Input/Output:

```
Enter price: nothing
error: unrecognized input
```

5. Number Ranges

Pitfall: Number Range

- **Range test**: decide if a number is between two numbers
  - min
  - max
- Math notation for x is between 0 and 10:

  $0 \le x \le 10$
- This **does not** work as expected in C++:

  ```
  if (0 <= x <= 10) {  // logic error
  ```
- Boolean expressions obey PEMDAS
- Evaluate left `<=`, and then right `<=`

Logic Error in Number Range

Per PEMDAS,

```
if (0 <= x <= 10) {
```

is evaluated like

```
if ((0 <= x) <= 10) {
```

1. `0 <= x` determines if x is non-negative; yields a bool
2. if is now like
   `if ((true/false) <= 10)`
3. Mixed expression, so `true/false` is implicitly promoted to `int 0/1`; if is like
   `if ((1/0) <= 10)`
4. Compare `1/0 <= 10`; this is **always true**

Correct Number Range

- Need to break `min <= x <= max` into two separate comparisons, ANDed together
- To test if variable *x* is between min and max (inclusive):

```
if ((min <= x) && (x <= max)) {
```

- Now
  - `(min <= x)` is evaluated first, produces `true/false`
  - `(x <= max)` is evaluated, produces `true/false`
  - && produces true only when both `(min <= x)` and `(x <= max)`
- Example:

```
if ((0 <= x) && (x <= 10)) {
```