Checking Multiple Conditions

- Conditional code often relies on a value being between two other values
- Consider these conditions:
  - Free shipping for orders over $25
  - 10 items or less
  - Children ages 3 to 11 allowed on play equipment
- What is the appropriate logical expression for each?

Checking Multiple Conditions

- Conditional code often relies on a value being between two other values
- Consider these conditions:
  - Free shipping for orders over $25

```
order_total > 25.0
```

  - 10 items or less

```
itemCount <= 10
```

  - Children ages 3 to 11 allowed on play equipment

```
(age >= 3) and (age <= 11)
```

- In the third example, we need to combine two logical expressions into one

Complex Logical Expressions

- Logical operators enable you to combine logical expressions
  - The result is a logical expression (evaluates to True or False)

```
(age >= 3) and (age < 11)
(age >= 3) or (age < 11)
not (age >= 3)
```

…which is the same as `(age < 3)`

- *and* and *or* are binary operators (2 operands)
  - and: True only if both operands are True
  - or: True if either operand is True, or both
- *not* is a unary operator (1 operand)
  - True if the operand is False, and vice versa

Car Insurance Example

- Determine the policy premium based on the following rules:

Gender	Age	Annual Premium
Male	Under 21	1500 + 200 for every ticket on record
Male	21 to 29	1200 + 100 for every ticket on record
Male	30 and older	1000 + 100 for every ticket on record
Female	Under 21	1200 + 200 for every ticket on record
Female	21 or older	1000 + 100 for every ticket on record

  - Five conditions with five outcomes

Design Questions

- What are the five conditions?
  - Are they mutually exclusive?
  - Do they cover all possibilities?
- What are the five outcomes that go along with those conditions?
- What variables (data) do the conditions and outcomes rely on?

From Design to Code

- Five conditions = five branches in an `if-else` tree
  - Five outcomes = the bodies of those branches

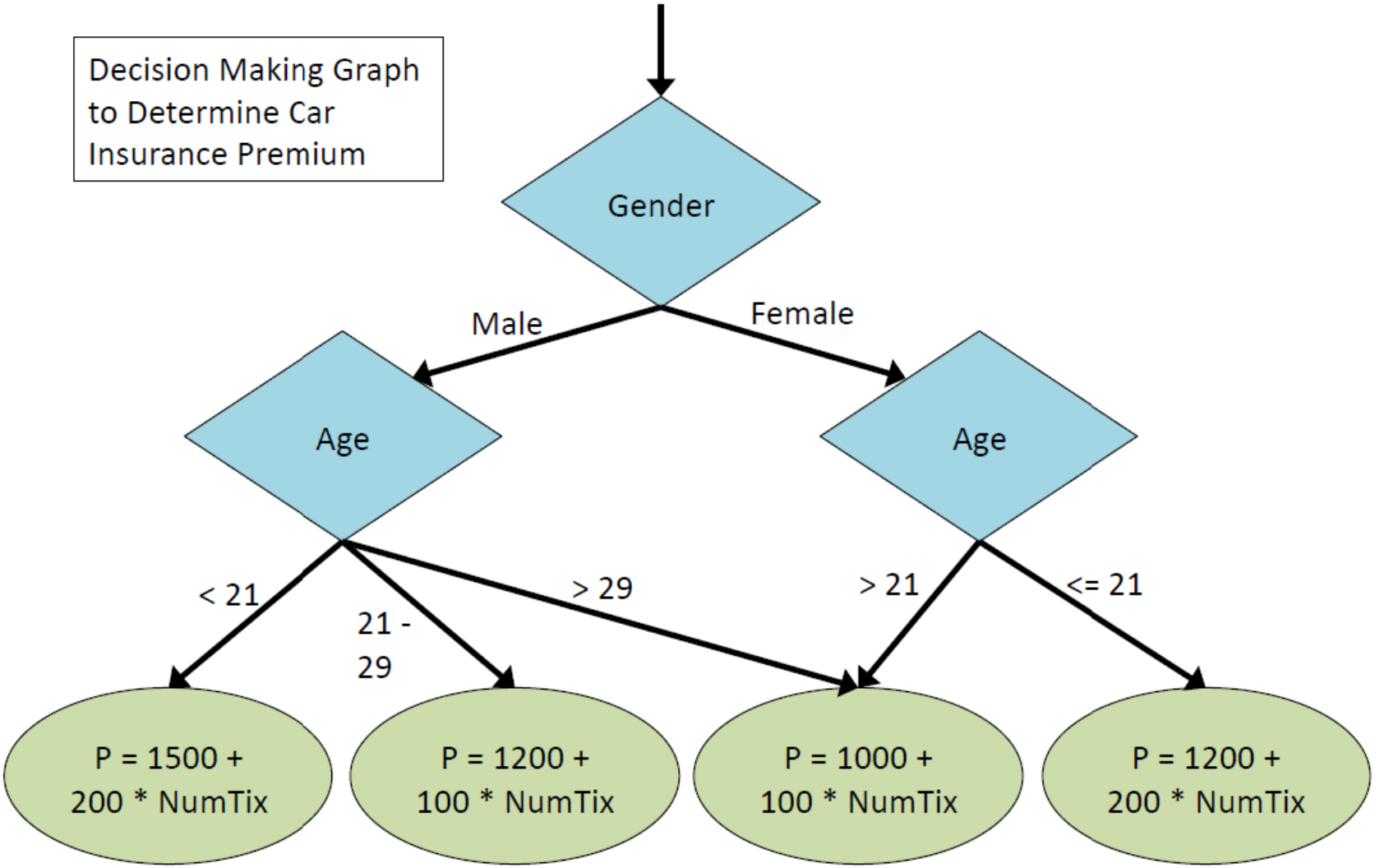
```
if (gender == 'M') and (age < 21):
   premium = 1500 + (200 * tickets)
elif (gender == 'M') and (age >=21) and (age < 29):
   premium = 1200 + (100 * tickets)
…
```

- Simplify! In the second branch, we already know that they aren't in the M under 21 group, so need to check again:

```
if (gender == 'M') and (age < 21):
   premium = 1500 + (200 * tickets)
elif (gender == 'M') ~~and (age >=21)~~ and (age < 29):
   premium = 1200 + (100 * tickets)
…
```

Alternate Design

- Instead of dividing the people into 5 groups:
  - Divide into two groups first (by gender)
  - Then divide those into groups (by age)


Decision Making Graph to Determine Car Insurance Premium
Gender
Male
Female
Age
Age
< 21
21 - 29
> 29
> 21
<= 21
P = 1500 + 200 * NumTix
P = 1200 + 100 * NumTix
P = 1000 + 100 * NumTix
P = 1200 + 200 * NumTix

Alternate Design

- Instead of dividing the people into 5 groups:
  - Divide into two groups first (by gender)
  - Then divide those into groups (by age)
- This results in a set of nested conditions
  - `gender == 'M'`
    - `age < 21`
    - `(age >= 21) and(age < 30)`
    - `age >= 30`
  - `gender == 'F'`
    - `age < 21`
    - `age >= 21`
  - Which can be converted into nested if-else trees

Range Checking

- Logical operators can be used to make arbitrarily complex expressions
- E.g.

`(x > 3) and (y <= 45) or (x == 15) and (y < x)` etc…

- But checking to see if a number is between two others is one of the most common

Order of Precedence

- Just like with arithmetic expressions
  - Evaluated from left to right
  - Arithmetic, then relational (comparisons), then logical
  - Parentheses can override precedence

Order of Precedence

- Just like with arithmetic expressions
  - Evaluated from left to right
  - Complete order of precedence (follows common sense)
    - *, /, //, %
    - +, -
    - <, <=, >=, >, ==, != (relational comparisons)
    - not
    - and
    - or
  - Parentheses can override precedence

Common Error

- The variable x is equal to 4 or 5:
  - Incorrect: `x == 4 or 5`
  - Correct: `(x == 4) or (x == 5)`

- The former sounds right in English, so is very common mistake
  - According to the order of precedence, it is evaluated as:

```
(x == 4) or 5
```

  - No matter what x is, this evaluates to 5
  - Any number that is not 0 is considered a True value

Common Error

- How would:

  ```
  x == (4 || 5)
  ```

  - Be evaluated?

Common Error: = vs. ==

- C++ allows you to use any expression that can be evaluated to either `true` or `false` as an expression in the `if` statement:

```
if (x = 5)
    cout << "The value is five." << endl;
```

- Very difficult mistake to catch
  - It is not a syntax error
  - It is a logical error

Floating-Point Equality

- Comparison of floating-point numbers for equality may not behave as you would expect
  - Example:
    - `1.0 == 3.0/7.0 + 2.0/7.0 + 2.0/7.0` evaluates to `False`
    - Why?

```
3.0/7.0 + 2.0/7.0 + 2.0/7.0 == 0.9999999999999989
```

- Solution: use a tolerance value
  - To compare x and y (using the build-in absolute value function):

```
abs(x - y) < 0.000001
```