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 a new operator to combine two logical expressions into one

Complex Logical Expressions

- Logical (Boolean) operators enable you to combine logical expressions

TABLE 4-5 Logical (Boolean) Operators in C++

Operator	Description
! ← unary	not
&& ← binary	and
|| ← binary	or

TABLE 4-7 The && (And) Operator

Expression1	Expression2	Expression1 && Expression2
`true` (nonzero)	`true` (nonzero)	`true` (1)
`true` (nonzero)	`false` (0)	`false` (0)
`false` (0)	`true` (nonzero)	`false` (0)
`false` (0)	`false` (0)	`false` (0)

EXAMPLE 4-3

Expression	Value	Explanation
`(14 >= 5) && ('A' < 'B')`	`true`	Because `(14 >= 5)` is `true`, `('A' < 'B')` is `true`, and `true && true` is `true`, the expression evaluates to `true`.
`(24 >= 35) && ('A' < 'B')`	`false`	Because `(24 >= 35)` is `false`, `('A' < 'B')` is `true`, and `false && true` is `false`, the expression evaluates to `false`.

TABLE 4-8 The || (Or) Operator

Expression1	Expression2	Expression1 || Expression2
`true` (nonzero)	`true` (nonzero)	`true` (1)
`true` (nonzero)	`false` (0)	`true` (1)
`false` (0)	`true` (nonzero)	`true` (1)
`false` (0)	`false` (0)	`false` (0)

EXAMPLE 4-4

Expression	Value	Explanation
`(14>=5) || ('A'>'B')`	`true`	Because `(14 >= 5)` is `true`, `('A' > 'B')` is `false`, and `true || false` is `true`, the expression evaluates to `true`.
`(24>=35) || ('A'>'B')`	`false`	Because `(24 >= 35)` is `false`, `('A' > 'B')` is `false`, and `false || false` is `false`, the expression evaluates to `false`.
`('A'<='a') || (7!=7)`	`true`	Because `('A' <= 'a')` is `true`, `(7 != 7)` is `false`, and `true || false` is `true`, the expression evaluates to `true`.

Logical (Boolean) Operators

TABLE 4-6 The ! (Not) Operator

Expression	!(Expression)
`true` (nonzero)	`false` (0)
`false` (0)	`true` (1)

EXAMPLE 4-2

Expression	Value	Explanation
`!('A' > 'B')`	`true`	Because `'A' > 'B'` is `false`, `!('A' > 'B')` is `true`.
`!(6 <= 7)`	`false`	Because `6 <= 7` is `true`, `!(6 <= 7)` is `false`.

Short-Circuit Evaluation

- Short-circuit evaluation: evaluation of a logical expression stops as soon as the value of the expression is known
- Example:

```
(age >= 21) || ( x == 5)      //Line 1
(grade == 'A') && (x >= 7)    //Line 2
```

Range Checking

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

```
(x > 3) && (y <= 45) || (x == 15) && (y < x)
```
etc…

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

Order of Precedence

- Relational and logical operators are evaluated from left to right
- The associativity is left to right
- Parentheses can override precedence

Order of Precedence (continued)

TABLE 4-9 Precedence of Operators

Operators	Precedence
!, +, – (unary operators)	first
*, /, %	second
+, –	third
<, <=, >=, >	fourth
==, !=	fifth
&&	sixth
||	seventh
= (assignment operator)	last

Order of Precedence (continued)

EXAMPLE 4-5

Suppose you have the following declarations:

```
bool found = true;
bool flag = false;
int num = 1;
double x = 5.2;
double y = 3.4;
int a = 5, b = 8;
int n = 20;
char ch = 'B';
```

Order of Precedence (continued)

Expression	Value	Explanation
`!found`	`false`	Because `found` is `true`, `!found` is `false`.
`x > 4.0`	`true`	Because `x` is `5.2` and `5.2 > 4.0` is `true`, the expression `x > 4.0` evaluates to `true`.
`!num`	`false`	Because `num` is 1, which is nonzero, `num` is `true` and so `!num` is `false`.
`!found && (x >= 0)`	`false`	In this expression, `!found` is `false`. Also, because `x` is `5.2` and `5.2 >= 0` is `true`, `x >= 0` is `true`. Therefore, the value of the expression `!found && (x >= 0)` is `false && true`, which evaluates to `false`.
`!(found && (x >= 0))`	`false`	In this expression, `found && (x >= 0)` is `true && true`, which evaluates to `true`. Therefore, the value of the expression `!(found && (x >= 0))` is `!true`, which evaluates to `false`.
`x + y <= 20.5`	`true`	Because `x + y = 5.2 + 3.4 = 8.6` and `8.6 <= 20.5`, it follows that `x + y <= 20.5` evaluates to `true`.

Order of Precedence (continued)

Expression	Value	Explanation
`(n >= 0) && (n <= 100)`	`true`	Here `n` is `20`. Because `20 >= 0` is `true`, `n >= 0` is `true`. Also, because `20 <= 100` is `true`, `n <= 100` is `true`. Therefore, the value of the expression `(n >= 0) && (n <= 100)` is `true && true`, which evaluates to `true`.
`('A' <= ch && ch <= 'Z')`	`true`	In this expression, the value of `ch` is `'B'`. Because `'A' <= 'B'` is `true`, `'A' <= ch` evaluates to `true`. Also, because `'B' <= 'Z'` is `true`, `ch <= 'Z'` evaluates to `true`. Therefore, the value of the expression `('A' <= ch && ch <= 'Z')` is `true && true`, which evaluates to `true`.
`(a + 2 <= b) && !flag`	`true`	Now `a + 2 = 5 + 2 = 7` and `b` is `8`. Because `7 <= 8` is `true`, the expression `a + 2 <= b` evaluates to `true`. Also, because `flag` is `false`, `!flag` is `true`. Therefore, the value of the expression `(a + 2 <= b) && !flag` is `true && true`, which evaluates to `true`.

Common Error

- The variable x is equal to 4 or 5:
  - Incorrect: `x == 4 || 5`
  - Correct: `(x == 4) || (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) || 5`
  - Which is always true (because `5` is always true)

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
  - Example: `fabs(x - y) < 0.000001`

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?

From Design to Code

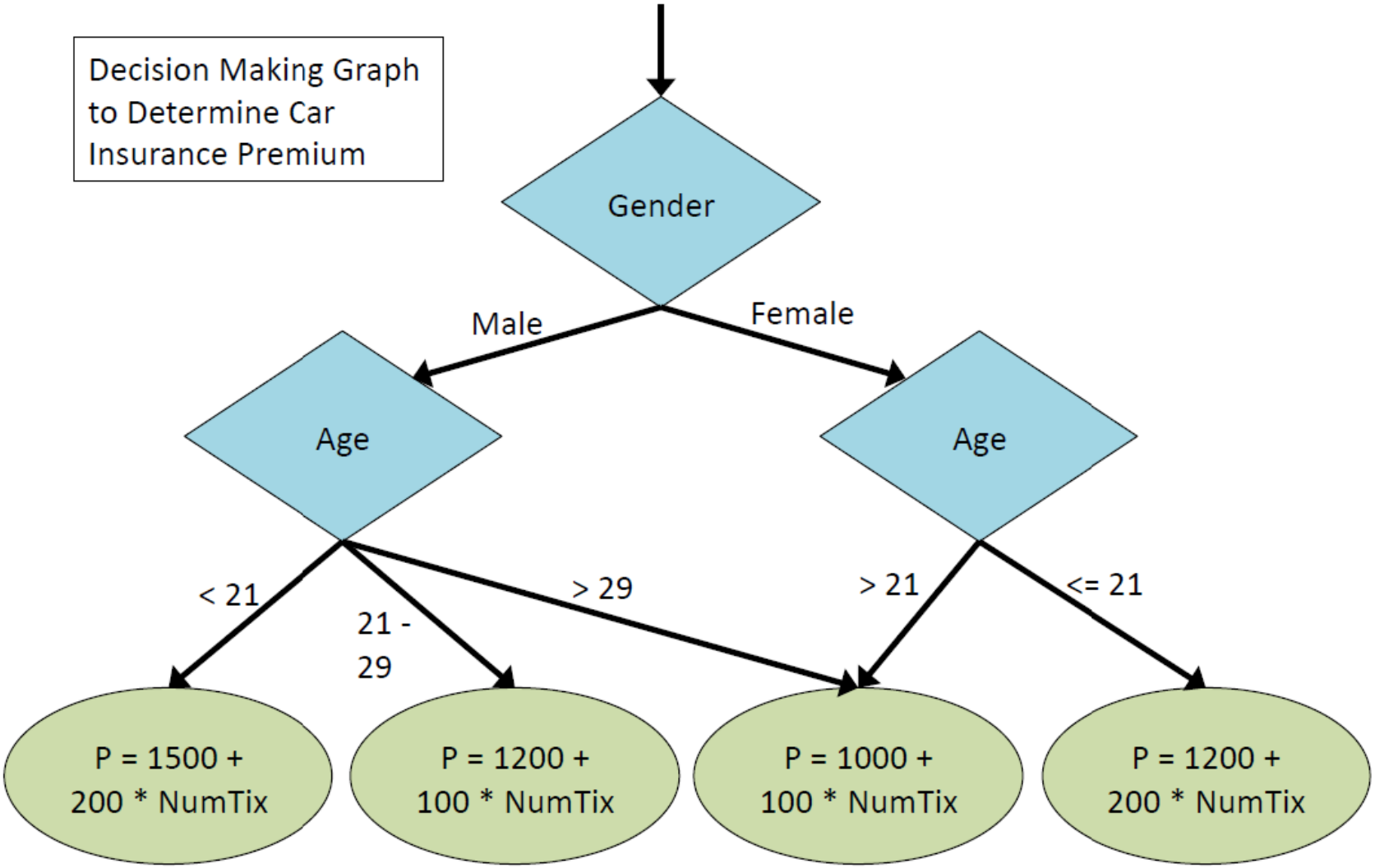
- Conditions and outcomes tell us what variables we need:
  - gender, age, premium, tickets

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

```
if( (gender == 'M') && (age < 21) )
{
   premium = 1500 + (200 * tickets);
}
else if( … )
…
```

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) && (age < 30)`
    - `age >= 30`
  - `gender == 'F'`
    - `age < 21`
    - `age >= 21`
  - Which can be converted into nested `if…else` trees


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