Decisions

CSE100 – Principles of Programming with C++
(based off Chapter 4 slides by Pearson)
Ryan Dougherty
Arizona State University
http://www.public.asu.edu/~redoughe/
Relational Operators

• Are used to compare numeric and `char` values to determine relative order

• Operators:

  >  Greater than
  <  Less than
  >= Greater than or equal to
  <= Less than or equal to
  == Equal to
  != Not equal to
Relational Expressions

• Relational expressions are Boolean (i.e., evaluate to true or false)
• Examples:
  12 > 5 is true
  7 <= 5 is false
  if x is 10, then
    x == 10 is true,
    x <= 8 is false,
    x != 8 is true, and
    x == 8 is false
Relational Expressions

• The value can be assigned to a variable
  ```
  bool result = (x <= y);
  ```
• Assigns 0 for `false`, 1 for `true`
• Do not confuse = (assignment) and == (equal to)
Use this when evaluating an expression that contains multiple relational operators.
if

• Supports the use of a decision structure, giving a program more than one path of execution
• Allows statements to be conditionally executed or skipped over
• It models the way we evaluate real-life situations
  “If it is cold outside, wear a coat and wear a hat.”
if (cont.)

```java
if (condition)
{
    statement1;
    statement2;
    ...
    statementn;
}
```

The block of statements inside the braces is called the **body** of the `if` statement. If there is only 1 statement in the body, the `{ }` may be omitted.
if (cont.)

• If \( \text{condition} \) is true, then the \textit{statement(s)} in the body are executed.

• If \( \text{condition} \) is false, then the \textit{statement(s)} are skipped.
Examples

    if (score >= 60)
        cout << "You passed." << endl;

    if (score >= 90)
    {
        grade = 'A';
        cout << "Wonderful job!" << endl;
    }
Key Notes

• **if** is a keyword. It must be lowercase

• *(condition)* must be in *( )*

• Do not place ; after *(condition)*

• Don't forget the { } around a multi-statement body

• Don’t confuse = *(assignment)* with == *(comparison)*
Comparison with Floating-Point

• It is difficult to test for equality when working with floating point numbers.

• It is better to use
  • greater-than or less-than tests, or
  • test to see if value is very close to a given value
What’s true/false?

• An expression whose value is 0 is considered **false**.
• An expression whose value is non-zero is considered **true**.
• An expression need not be a comparison – it can be a single variable or a mathematical expression.
• A **flag** is a variable that signals a condition
• It is usually implemented as a **bool**
• Meaning:
  • **true**: the condition exists
  • **false**: the condition does not exist
• The flag value can be both set and tested with **if** statements
Flag Example

```cpp
bool validMonths = true;
...
if (months < 0)
    validMonths = false;
...
if (validMonths)
    monthlyPayment = total / months;
```
Integer Flags

• Integer variables can be used as flags

• Remember that 0 means false, any other value means true

  int allDone = 0;  // set to false
  ...  
  if (count > MAX_STUDENTS)
     allDone = 1;  // set to true
  ...  
  if (allDone)
     cout << "Task finished";
if/else

• Allows a choice between statements depending on whether (condition) is true or false

• Format:  
  
  ```
  if (condition)
  {
    statement set 1;
  }
  else
  {
    statement set 2;
  }
  ```
if/else (cont.)

• If \( (\text{condition}) \) is true, \textit{statement set 1} is executed and \textit{statement set 2} is skipped.

• If \( (\text{condition}) \) is false, \textit{statement set 1} is skipped and \textit{statement set 2} is executed.
if/else Example

```cpp
if (score >= 60)
    cout << "You passed.\n";
else
    cout << "You did not pass.\n";

if (intRate > 0)
{  interest = loanAmt * intRate;
    cout << interest;
}
else
    cout << "You owe no interest.\n";
```
When to use just if?

If there are two conditions and both of them can be true or both can be false, then use two `if` statements:

```cpp
if (num > 0)
    cout << num << " is positive\n";
if (num %2 == 0)
    cout << num << " is even\n";
```

If the two conditions cannot both be true, then a single `if/else` statement can work:

```cpp
if (num %2 == 0)
    cout << num << " is even\n";
else
    cout << num << " is odd\n";
```
if/else Chains

• This is a chain of `if` statements that test in order until one is found to be true
• This also models thought processes
  “If it is raining, take an umbrella, else, if it is windy, take a hat, else, if it is sunny, take sunglasses.”
Trailing else

• Is used with a set of `if/else if` statements

• It provides a default statement or action that is performed when none of the conditions is true

• It can be used to catch invalid values or handle other exceptional situations

```cpp
if (age >= 21)
    cout << "Adult";
else if (age >= 13)
    cout << "Teen";
else if (age >= 2)
    cout << "Child";
else
    cout << "Baby";
```
Nested if Statements

• An if statement that is part of the if or else part of another if statement

• This can be used to evaluate > 1 data item or to test > 1 condition

```java
if (score < 100)
{
    if (score > 90)
        grade = 'A';
}
```
Key Note: Nested if Statements

• An **else** matches the nearest previous **if** that does not have an **else**

```java
if (score < 100)
    if (score > 90)
        grade = 'A';
    else ...  // goes with second if,
              // not first one
```

• Proper indentation aids understanding
Example Program: Menu

• **Menu**: list of choices presented to the user on the computer screen
• **Menu-driven program**: program execution is controlled by user selecting from a list of actions
• A menu-driven program can be written using `if/else if` statements
• Display a list of numbered or lettered choices for actions.
• Input user’s selection of number or letter
• Test the user selection in `(condition)`
  • if a match, then execute code to carry out desired action
  • if not, then test with next `(condition)`
Logical Operators

- Are used to create relational expressions from other relational expressions

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;&amp;</td>
<td><strong>AND</strong></td>
<td>New relational expression is true if both expressions are true</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>!</td>
<td><strong>NOT</strong></td>
<td>Reverses the value of an expression; true expression becomes false, false expression becomes true</td>
</tr>
</tbody>
</table>
# Logical Operator Examples

```c
int x = 12, y = 5, z = -4;
```

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>(x &gt; y) &amp;&amp; (y &gt; z)</code></td>
<td>true or 1</td>
</tr>
<tr>
<td><code>(x &gt; y) &amp;&amp; (z &gt; y)</code></td>
<td>false or 0</td>
</tr>
<tr>
<td>`(x &lt;= z)</td>
<td></td>
</tr>
<tr>
<td>`(x &lt;= z)</td>
<td></td>
</tr>
<tr>
<td>!(x &gt;= z)</td>
<td>false</td>
</tr>
</tbody>
</table>
Logical Operators and bools

• Logical operators can be used with `bool` variables as well as expressions that evaluate to `true` or `false`.

• Ex:
  ```
  bool done = false;
  if ( (!done) && (count < 6) )
  {
      
  }
  ```
Short-Circuit Evaluation

• If an expression using the `&&` operator is being evaluated and the subexpression on the left side is `false`, then there is no reason to evaluate the subexpression on the right side. It is skipped.

• If an expression using the `||` operator is being evaluated and the subexpression on the left side is `true`, then there is no reason to evaluate the subexpression on the right side. It is skipped.
Logical Operator Precedence

Highest       \\
               &&

Lowest        ||

Example:
(2 < 3) || (5 > 6) && (7 > 8)

is true because AND is evaluated before OR
More on Precedence

<table>
<thead>
<tr>
<th>Highest</th>
<th>arithmetic operators</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>relational operators</td>
</tr>
<tr>
<td>Lowest</td>
<td>logical operators</td>
</tr>
</tbody>
</table>

Example:

8 < 2 + 7 || 5 == 6 is true
Example: Check Valid Range

• Used to test if a value is within a range
  `if (grade >= 0 && grade <= 100)
     cout << "Valid grade";`

• You can also test if a value lies outside a range
  `if (grade <= 0 || grade >= 100)
     cout << "Invalid grade";`

• Note that you cannot use mathematical notation
  `if (0 <= grade <= 100) //Doesn’t
    //work!`
User Input Validation

• **Input validation**: inspecting input data to determine if it is acceptable
• You want to avoid accepting bad input
• You can perform various tests
  • Range
  • Reasonableness
  • Valid menu choice
  • Zero as a divisor
Scope

• The **Scope** of a variable is the block in which it is defined, from the point of definition to the end of the block

• Variables are usually defined at the beginning of a function

• They may instead be defined close to the place where they are first used

• Variables defined inside `{ }` have **local** or **block scope**

• When in a block that is nested inside another block, you can define variables with the same name as in the outer block.
  • When the program is executing in the inner block, the outer definition is not available
  • This generally not a good idea. The program may be hard to read or understand.
Characters/Strings and if

• You can use relational operators with characters and string objects

if (menuChoice == 'A')
if (name1 >= name2)

• Comparing characters is really comparing the ASCII values of characters

• Comparing string objects is comparing the ASCII values of the characters in the strings. Comparison is character-by-character, starting with the first character of each string.

• You cannot compare C-style strings with relational operators
Testing Characters

• These functions require the `ctype` header file

<table>
<thead>
<tr>
<th>FUNCTION</th>
<th>MEANING</th>
</tr>
</thead>
<tbody>
<tr>
<td>isalpha</td>
<td>true if arg. is a letter, false otherwise</td>
</tr>
<tr>
<td>isalnum</td>
<td>true if arg. is a letter or digit, false otherwise</td>
</tr>
<tr>
<td>isdigit</td>
<td>true if arg. is a digit 0-9, false otherwise</td>
</tr>
<tr>
<td>islower</td>
<td>true if arg. is lowercase letter, false otherwise</td>
</tr>
<tr>
<td>isprint</td>
<td>true if arg. is a printable character, false otherwise</td>
</tr>
<tr>
<td>ispunct</td>
<td>true if arg. is a punctuation character, false otherwise</td>
</tr>
<tr>
<td>isupper</td>
<td>true if arg. is an uppercase letter, false otherwise</td>
</tr>
<tr>
<td>isspace</td>
<td>true if arg. is a whitespace character, false otherwise</td>
</tr>
</tbody>
</table>
Switch

• Is uses the value of an integer expression to determine the statements to execute

• It may sometimes be used instead of `if/else if` statements

```java
switch (IntExpression)
{
    case exp1: statement set 1;
    case exp2: statement set 2;
    ...
    case expn: statement set n;
    default: statement set n+1;
}
```
Switch Requirements

1) *IntExpression* must be an integer variable or a *char*, or an expression that evaluates to an integer value

2) *exp1* through *expn* must be constant integer type expressions and must be unique in the *switch* statement

3) *default* is optional but recommended
How Switch Works

1) \textit{IntExpression} is evaluated

2) The value of \textit{intExpression} is compared against \textit{exp1} through \textit{expn}.

3) If \textit{IntExpression} matches value \textit{expi}, the program branches to the statement(s) following \textit{expi} and continues to the end of the \textit{switch}

4) If no matching value is found, the program branches to the statement after \textit{default}:
break

- Is used to stop execution in the current block
- It is also used to exit a `switch` statement
- It is used to execute a single `case` statement without executing statements following it

```cpp
switch (gender)
{
    case 'f': cout << "female";
        break;
    case 'm': cout << "male";
        break;
    default : cout << "invalid gender";
}
```
Switch with Menu

A `switch` statement is a natural choice for a menu-driven program

- display menu
- get user input
- use user input as `IntExpression` in `switch` statement
- use menu choices as `exp` values to test against in the `case` statements
Enums

• Is a data type created by the programmer
• Contains a set of named integer constants
• Format:
  
  ```
  enum name {val1, val2, ... valn};
  ```
  
• Examples:
  
  ```
  enum Fruit {apple, grape, orange};
  enum Days {Mon, Tue, Wed, Thur, Fri};
  ```
Enumerated Variables

• To define variables, use the enumerated data type name

```java
Fruit snack;
Days workDay, vacationDay;
```

• A variable may contain any valid value for the data type

```java
snack = orange;      // no quotes
if (workDay == Wed)   // none here
```

• Enumerated data type values are associated with integers, starting at 0

```java
enum Fruit {apple, grape, orange};
```

```
<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>apple</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>grape</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>orange</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

• You can override the default association

```java
enum Fruit {apple = 2, grape = 4, orange = 5};
```
Enumerated Values

- Enumerated data type values can be compared using their integer values
  
  ```java
  if (snack == 1)
  ```

- Enumerated data type values cannot be assigned using their integer values
  
  ```java
  snack = 2;  // won't work
  ```
Enum Notes

• Enumerated data types improve the readability of a program
• Enumerated variables can not be used with input statements, such as `cin`
• An enumerated variable will not display the name associated with the value of an enumerated data type if used with `cout`