Using Boolean Expressions

- A Boolean Expression is an expression that is either true or false
  - Boolean expressions are evaluated using relational operations such as
    - \( = \), \(<\), and \(\geq\) which produce a boolean value
  - and boolean operations such as
    - \(&&\), \(|\|\), and \(!\) which also produce a boolean value
- Type bool allows declaration of variables that carry the value true or false

Evaluating Boolean Expressions

- Boolean expressions are evaluated using values from the Truth Tables in
  - For example, if \(y\) is 8, the expression
    \(\neg ( (y < 3) \| (y > 7))\)
  is evaluated in the following sequence
    \(\neg (\text{false} \| \text{true})
    \neg (\text{true})
    \text{false}

Order of Precedence

- If parenthesis are omitted from boolean expressions, the default precedence of operations is:
  - Perform \(!\) operations first
  - Perform relational operations such as \(<\) next
  - Perform \(&&\) operations next
  - Perform \(|\|\) operations last
### Precedence Rules

- Items in expressions are grouped by precedence rules for arithmetic and boolean operators.
  - Operators with higher precedence are performed first.
  - Binary operators with equal precedence are performed left to right.
  - Unary operators of equal precedence are performed right to left.

### Precedence Rule Example

- The expression 
  \[(x+1) > 2 \quad \text{||} \quad (x + 1) < -3\]
  is equivalent to
  \[( (x + 1) > 2) \quad \text{||} \quad ( (x + 1) < -3)\]
  - Because > and < have higher precedence than ||
- and is also equivalent to
  \[x + 1 > 2 \quad \text{||} \quad x + 1 < -3\]
Evaluating

- \( x + 1 > 2 \) \( \| \) \( x + 1 < -3 \)

- Using the precedence rules of from the previous table
  - First apply the unary \(-\) 
  - Next apply the '+'s 
  - Now apply the '>' and '<' 
  - Finally do the ' | ' 

Short-Circuit Evaluation

- Some boolean expressions do not need to be completely evaluated
  - if \( x \) is negative, the value of the expression \( (x >= 0) \) \&\& \( (y > 1) \) can be determined by evaluating only \( x >= 0 \)

- C++ uses short-circuit evaluation
  - If the value of the leftmost sub-expression determines the final value of the expression, the rest of the expression is not evaluated

Using Short-Circuit Evaluation

- Short-circuit evaluation can be used to prevent run time errors
  - Consider this if-statement

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

- If the value of kids is zero, short-circuit evaluation prevents evaluation of \( (pieces / 0 >= 2) \)
  - Division by zero causes a run-time error

Type bool and Type int

- C++ can use integers as if they were Boolean values
  - Any non-zero number (typically 1) is true
  - 0 (zero) is false
Problems with !

• The expression \( ! \text{time} > \text{limit} \), with limit = 60, is evaluated as \((!\text{time}) > \text{limit}\)
• If time is an int with value 36, what is !time?
  – False! Or zero since it will be compared to an integer
  – The expression is further evaluated as \(0 > \text{limit}\)  
    false

Correcting the ! Problem

• The intent of the previous expression was most likely the expression
  
  \((! (\text{time} > \text{limit}))\)

  which evaluates as
  
  \((! (\text{false}))\)  
  
  true

Avoiding !

• Just as not in English can make things not undifficult to read, the ! operator can make C++ expressions difficult to understand
• Before using the ! operator see if you can express the same idea more clearly without the ! operator

bool Return Values

• A function can return a bool value
  – Such a function can be used where a boolean expression is expected
    • Makes programs easier to read
• if \(((\text{rate} \geq 10) \&\& (\text{rate} < 20)) \| (\text{rate} == 0))\) is easier to read as
  
  if (appropriate (\text{rate}))
  
  – If function appropriate returns a bool value based on the expression above
Multiway Branches

- A branching mechanism selects one out of a number of alternative actions
  - The if-else-statement is a branching mechanism
- Branching mechanisms can be a subpart of another branching mechanism
  - An if-else-statement can include another if-else-statement as a subpart

Nested Statements

- A statement that is a subpart of another statement is a nested statement
  - When writing nested statements it is normal to indent each level of nesting
  - Example:

```cpp
if (count < 10)
  if (x < y)
    cout << x << " is less than " << y;
  else
    cout << y << " is less than " << x;
```

Nested if-else Statements

- Use care in nesting if-else-statements
- Example: To design an if-else statement to warn a driver when fuel is low, but tells the driver to bypass pit stops if the fuel is close to full. Otherwise there should be no output.

Pseudocode: if fuel gauge is below ¾ then:
  if fuel gauge is below ¼ then:
    issue a warning
  otherwise (gauge > ¾) then:
    output a statement saying don’t stop
First Try Nested if's

• Translating the previous pseudocode to C++ could yield (if we are not careful)
  
  if (fuel_gauge_reading < 0.75)
  
  if (fuel_gauge_reading < 0.25)
  
  cout << "Fuel very low. Caution!\n";
  
  else
  
  cout << "Fuel over 3/4. Don't stop now!\n";
  
  – This would compile and run, but does not produce the desired results
  
  – The compiler pairs the "else" with the nearest previous "if"

Braces and Nested Statements

• Braces in nested statements are like parenthesis in arithmetic expressions
  
  – Braces tell the compiler how to group things
  
• Use braces around substatements
  
• Consider the next example that demonstrates the use of braces in nested if-else-statements.

Multi-way if-else-statements

• An if-else-statement is a two-way branch
  
• Three or four (or more) way branches can be designed using nested if-else-statements
  
  – Example: The number guessing game with the number stored in variable number, the guess in variable guess. How do we give hints?
Number Guessing

• The following nested statements implement the hints for our number guessing game
  – if (guess > number)
    cout << "Too high."
  else
    if (guess < number)
      cout << "Too low."
    else
      if (guess == number)
        cout << "Correct!";

Indenting Nested if-else

• Notice how the code on the previous slide crept across the page leaving less and less space
  – Use this alternative for indenting several nested if-else-statements:
    if (guess > number)
      cout << "Too high."
    else if (guess < number)
      cout << "Too low."
    else if (guess == number)
      cout << "Correct!";

The Final if-else-statement

• When the conditions tested in an if-else-statement are mutually exclusive, the final if-else can sometimes be omitted.
  – The previous example can be written as
    if (guess > number)
      cout << "Too high."
    else if (guess < number)
      cout << "Too low."
    else // (guess == number)
      cout << "Correct!";

Nested if-else Syntax

• A Multiway if-else statement is written as
  – if(Boolean_Expression_1)
    Statement_1
  else if (Boolean_Expression_2)
    Statement_2
  …
  else if (Boolean_Expression_n)
    Statement_n
  else
    Statement_For_All_Other_Possibilities
Example: State Income Tax

- Write a program for a state that computes tax according to the rate schedule:

  No tax on first $15,000 of income

  5% tax on each dollar from $15,001 to $25,000

  10% tax on each dollar over $25,000

Refining if-else-statements

- Notice that the line
  ```
  else if (( net_income > 15000
             && net_income <= 25000))
  ```
  can be replaced with

  ```
  else if (net_income <= 25000)
  ```

  - The computer will not get to this line unless it is already determined that net_income > 15000

If-else statement

```c
if (net_income <= 15000)
    return 0;
else if (((net_income > 15000) && (net_income <= 25000)))
    //return 5% of amount over $15,000
    return (0.05*(net_income - 15000));
else if (net_income > 25000)
{  //five_percent_tax = 5% of income from $15,000 to $25,000.
    five_percent_tax = 0.05*10000;
    //ten_percent_tax = 10% of income over $25,000.
    ten_percent_tax = 0.10*(net_income - 25000);
    return (five_percent_tax + ten_percent_tax);
}
```

The switch-statement

- The switch-statement is an alternative for constructing multi-way branches
  - The next example determines output based on a letter grade
    - Grades 'A', 'B', and 'C' each have a branch
    - Grades 'D' and 'F' use the same branch
    - If an invalid grade is entered, a default branch is used
Multi-way Selections – Switch

- The switch statement is a multi-way decision that tests whether an expression matches a number of constant integer values and branches accordingly:

```
switch (expression)
{
    case const-expr: statements
        break;
    case const-expr: statements
        break;
    default: statements
        break;
}
```

switch-statement Syntax

- switch (controlling expression)
  {
    case Constant_1:
        statement_Sequence_1
        break;
    case Constant_2:
        Statement_Sequence_2
        break;
    . . .
    case Constant_n:
        Statement_Sequence_n
        break;
    default: Default_Statement_Sequence
  }

The Controlling Statement

- A switch statement's controlling statement must return one of these types
  - A bool value
  - An enum constant
  - An integer type
  - A character
- The value returned is compared to the constant values after each "case"
  - When a match is found, the code for that case is used
The break Statement

- The break statement ends the switch-statement
  - Omitting the break statement will cause the code for the next case to be executed!
  - Omitting a break statement allows the use of multiple case labels for a section of code
    - case 'A':
      case 'a':
        cout << "Excellent."
        break;
    - Runs the same code for either 'A' or 'a'

The default Statement

- If no case label has a constant that matches the controlling expression, the statements following the default label are executed
  - If there is no default label, nothing happens when the switch statement is executed
  - It is a good idea to include a default section

Switch-statements and Menus

- Nested if-else statements are more versatile than a switch statement
- Switch-statements can make some code more clear
  - A menu is a natural application for a switch-statement

Sample Dialogue

Choose 1 to see the next homework assignment.
Choose 2 for your grade on the last assignment.
Choose 3 for assignment hints.
Choose 4 to exit this program.

Enter your choice and press Return: 3

Assignment Hints:
Analyze the problem.
Write an algorithm in pseudocode.
Translate the pseudocode into a C++ program.

Choose 1 to see the next homework assignment.
Choose 2 for your grade on the last assignment.
Choose 3 for assignment hints.
Choose 4 to exit this program.

Enter your choice and press Return: 4

End of Program.