Objectives

- Understand Boolean expressions;
- Branching in a program using if and Boolean expressions;
- Using while and do-while to implement looping structures;
- Calculating sums and products.

Boolean Expressions

- Recall that when treating any quantity as Boolean in C++, a value of zero translates to false, and any other value translates to true.
- There are two Boolean constants (actually reserved words), and they are true and false.
- You can create Boolean expressions using comparison operators.

Expressions from Operators

- <
- >
- >=
- <=
- == This a double equal sign. Do not confuse this with the replacement operator =.
- !=
Expressions from Operators (cont.)

- $x < y$ true if $x$ is less than $y$
- $x \geq y$ true if $x$ is greater or equal to $y$
- $x \neq y$ true if $x$ and $y$ are not equal
- $x > y$ true if $x$ is greater than $y$
- $x == y$ true if $x$ and $y$ are the same (All bits in their respective representations must be the same.)

Building More Complex Expressions

There are logical operators that can be used to combine logical expressions or negate an expression.

- $||$ The "or" operator. (binary)
- $&&$ The "and" operator (binary)
- $!$ The "not" operator (unary)

Binary requires two operands, and unary requires one operand.

$||$ (or)

$(x<y) || (z > w)$ True if either $x$ is less than $y$ or $z$ is greater than $w$ or both are true.

| Exp1 | Exp2 | Exp1 $||$ Exp2 |
|------|------|----------------|
| true | true | true           |
| true | false| true           |
| false| true | true           |
| false| false| false          |
&& (and)

(x<y) && (z > w) True only if x is less than y and z is greater than w

<table>
<thead>
<tr>
<th>Exp1</th>
<th>Exp2</th>
<th>Exp1 &amp;&amp; Exp2</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
</tbody>
</table>

! (not)

!(x > y) True if x is not greater than y, or in other words, x is less than or equal to y.

<table>
<thead>
<tr>
<th>Expression</th>
<th>!Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
</tr>
</tbody>
</table>

Branching using if

if (expression) {
    series of statements
}

If the expression is true, the code inside the {} is executed; otherwise, it is skipped. If the {} are missing, the if applies only to the next single statement. You must keep this in mind as you design your programs.
Branching using **if-else**

```c++
if (expression) {
    code set 1
} else {
    code set 2
}
```

Exactly one of the two code sets will be executed; set 1 if expression is true; otherwise set 2.

**if – else - if**

This structure can be used to test multiple conditions, often in a very beneficial way.

What is the example on the next slide designed to do?

```c++
cin >> avg;
if(avg >= 90)
    cout << "Excellent\n";
else
    if(avg >= 70)
        cout << "Average\n";
    else
        if(avg >= 60)
            cout << "Passing\n";
        else
            cout << "Try harder\n";
```
Trouble!!

Consider the following code segment:
```
int x = 4;
if(x = 5) //Note the single =
    cout << "The moon is made of green cheese\n";
else
    cout << "The moon has no atmosphere\n";
What will happen? Is the moon made of green cheese, or does it have no atmosphere?
```

Looping using while
```
while(expression){
    code
}
Test the expression, and if it is true, execute the code inside {}. When execution of the code is complete, test the expression again, executing the code if it is true. Continue until the expression becomes false.
Note: This is a pre-test loop structure in that the evaluation of the expression comes before the block of code.
What happens if the code inside {} is executed but never changes the value of the expression?
```

while (cont.)
```
If the {} associated with a while are omitted, the structure involves only the next single statement.
```
while(x < y)
x = x+1 ;
```
**while Trouble!!**

```cpp
while (x < y)
    x = x + 1;
    cout << x << endl;

What happens here?
```

**More while Trouble!!**

```cpp
while (x < y)
    cout << x << endl;
    x = x + y;

What happens here?
```

**More while Trouble!!**

```cpp
while (x < y);  // Note the semicolon.
{
    cout << x << endl;
    x = x + y;
}
What happens here?
```
Loop Forever

What happens in the following?
while(true){
    code
}
It would loop forever, unless (code) provides for an exit from within the loop. It is possible to do multiple tests from multiple places and exit when any of the conditions tested occurs. Exit a loop in this manner by executing a break instruction. For example,
    if (x < y) break;
Exiting a loop from within can complicate the logic and diminish the readability of the program. Use this feature with care.

You Try It

Working in a group write a program that will:
- Request the input of three positive integers;
- Test those three integers to see if they are valid lengths of the sides of a non-degenerate triangle.
- If they are, output the result. If they aren’t, indicate so and say why. Examples:
  3, 4, and 5 are valid lengths.
  3, 4, and 8 are not valid since 8 >= 4 + 3.
This is readily done with if-else-if. When designing the logic for this, assume that only positive numbers are input.

Change the Previous Program

Once the program from the previous slide is working, change it so that it continues to request three numbers and report the results each time until all three numbers input have value 10.