2.4

Simple Flow of Control
Simple Flow of Control

- Flow of control
  - The order in which statements are executed

- Branch
  - Lets program choose between two alternatives

Branch Example

- To calculate hourly wages there are two choices
  - Regular time (up to 40 hours)
    \[
    \text{gross\_pay} = \text{rate} \times \text{hours};
    \]
  - Overtime (over 40 hours)
    \[
    \text{gross\_pay} = \text{rate} \times 40 + 1.5 \times \text{rate} \times (\text{hours} - 40);
    \]
  - The program must choose which of these expressions to use
Designing the Branch

• Decide if (hours >40) is true
  – If it is true, then use
    \[ \text{gross\_pay} = \text{rate} \times 40 + 1.5 \times \text{rate} \times (\text{hours} - 40); \]
  – If it is not true, then use
    \[ \text{gross\_pay} = \text{rate} \times \text{hours}; \]

Implementing the Branch

• if-else statement is used in C++ to perform a branch

```cpp
if (hours > 40)
    \text{gross\_pay} = \text{rate} \times 40 + 1.5 \times \text{rate} \times (\text{hours} - 40);
else
    \text{gross\_pay} = \text{rate} \times \text{hours};
```
An *if-else* Statement

```cpp
#include <iostream>
#include <string>
using namespace std;
int main()
{
    float Hours;
    double Gross_pay, rate;
    cin >> Hours;
    cout << "Enter the hourly rate of pay: ";
    cin >> rate;
    cout << "Enter the number of hours worked, in 
" << "rounded to a whole number of hours: ";
    cin >> Hours;
    if (Hours > 40)
        Gross_pay = rate*40 + 1.5*rate*(Hours - 40);
    else
        Gross_pay = rate*Hours;
    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    cout << "Hourly pay rate = 
" << "hours <= end);"
    cout << "Hourly pay rate = 
" << Gross_pay << endl;
    cout << "Gross pay = 
" << Gross_pay << endl;
    return 0;
}
```

**Sample Dialogue 1**

Enter the hourly rate of pay: 20.00
Enter the number of hours worked, rounded to a whole number of hours: 30
Gross pay = 600.00

**Sample Dialogue 2**

Enter the hourly rate of pay: 30.00
Enter the number of hours worked, rounded to a whole number of hours: 41
Gross pay = 1234.00

---

Syntax for an *if-else* Statement

**A Single Statement for Each Alternative:**

```
if (Boolean_Expression)
    Yes_Statement
else
    No_Statement
```

**A Sequence of Statements for Each Alternative:**

```
if (Boolean_Expression)
{
    Yes_Statement_1
    Yes_Statement_2
    ...
    Yes_Statement_Last
}
else
{
    No_Statement_1
    No_Statement_2
    ...
    No_Statement_Last
}
```
### if-else Flow Control (1)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>if (boolean expression)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>true statement</td>
</tr>
<tr>
<td>else</td>
<td></td>
</tr>
<tr>
<td></td>
<td>false statement</td>
</tr>
</tbody>
</table>

- When the boolean expression is true
  - Only the true statement is executed

- When the boolean expression is false
  - Only the false statement is executed

### if-else Flow Control (2)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>if (boolean expression)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>true statements</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td>else</td>
<td></td>
</tr>
<tr>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>false statements</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
</tbody>
</table>

- When the boolean expression is true
  - Only the true statements enclosed in { } are executed

- When the boolean expression is false
  - Only the false statements enclosed in { } are executed
Compound Statements

- A compound statement is more than one statement enclosed in {}.

- Branches of if-else statements often need to execute more than one statement.

- Example:

```java
if (boolean expression)
{
    true statements
}
else
{
    false statements
}
```

---

**Compound Statements Used with if-else**

```java
if (my_score > your_score)
{
    cout << "I win!\n";
    wager = wager + 100;
}
else
{
    cout << "I wish these were golf scores.\n";
    wager = 0;
}
```
Boolean Expressions

- Boolean expressions are expressions that are either true or false.

- Comparison operators such as '>' (greater than) are used to compare variables and/or numbers.
  
  - (hours > 40) Including the parentheses, is the boolean expression from the wages example.
  
  - A few of the comparison operators that use two symbols (No spaces allowed between the symbols!)
    
    - >= greater than or equal to
    - != not equal or inequality
    - == equal or equivalent

---

Comparison Operators

<table>
<thead>
<tr>
<th>Math Symbol</th>
<th>English</th>
<th>C++ Notation</th>
<th>C++ Sample</th>
<th>Math Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>equal to</td>
<td>==</td>
<td>x + 7 == 2 * y</td>
<td>x + 7 = 2y</td>
</tr>
<tr>
<td>!=</td>
<td>not equal to</td>
<td>!=</td>
<td>ans != 'n'</td>
<td>ans != 'n'</td>
</tr>
<tr>
<td>&lt;</td>
<td>less than</td>
<td>&lt;</td>
<td>count &lt; m + 3</td>
<td>count &lt; m + 3</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or</td>
<td>&lt;=</td>
<td>time &lt;= limit</td>
<td>time ≤ limit</td>
</tr>
<tr>
<td></td>
<td>equal to</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt;</td>
<td>greater than</td>
<td>&gt;</td>
<td>time &gt; limit</td>
<td>time &gt; limit</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than</td>
<td>&gt;=</td>
<td>age &gt;= 21</td>
<td>age ≥ 21</td>
</tr>
<tr>
<td></td>
<td>or equal to</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
AND operator: &&

- Boolean expressions can be combined into more complex expressions with
  - &&
  - The AND operator
    - And expression is true if both sub-expressions are true

- Syntax: (Comparison_1) && (Comparison_2)

- Example: if ( (2 < x) && (x < 7) )
  - True only if x is between 2 and 7
  - Inside parentheses are optional but enhance meaning

OR operator: ||

- ||
- The OR operator (no space!)
  - Or expression is true if either or both sub-expressions are true

- Syntax: (Comparison_1) || (Comparison_2)

- Example: if ( ( x = = 1) || ( x = = y) )
  - True if x contains 1
  - True if x contains the same value as y
  - True if both comparisons are true
NOT operator: !

- !
- negates any boolean expression
  - !( x < y)
    - True if x is NOT less than y
  - !(x == y)
    - True if x is NOT equal to y

- ! Operator can make expressions difficult to understand… use only when appropriate

3.1

Using Boolean Expressions
Inequalities

• Be careful translating inequalities to C++
• if $x < y < z$ translates as

    if ( ( $x < y$ ) && ( $y < z$ ) )

    NOT

    if ( $x < y < z$ )
Pitfall: Using = or ==

- =
  - the assignment operator
    - Used to assign values to variables
    - Example: \( x = 3; \)

- ==
  - the equality operator
    - Used to compare values
    - Example: \( \text{if } (x == 3) \)
  - The compiler will accept this error:
    \( \text{if } (x = 3) \)
    - but stores 3 in x instead of comparing x and 3
    - Since the result is 3 (non-zero), the expression is true

Boolean Expressions: Summary

- A Boolean Expression is an expression that is either true or false
  - Boolean expressions are evaluated using relational operations such as
    - = =, <, and >= 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
    
    \[
    ! ( ( y < 3 ) || ( y > 7 ) )
    \]

    is evaluated in the following sequence:
    - \((y<3)\) is evaluated and evaluates to false,
    - then, \((y>7)\) is evaluated and evaluates to true
    - then \(!true\) is evaluated and evaluates to 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
  
  $$(x + 1) > 2 \ || \ (x + 1) < -3$$

  is equivalent to
  
  $$((x + 1) > 2) \ || \ ((x + 1) < -3)$$

  - Because $>$ and $<$ have higher precedence than $\|$ $\|$

- and is also equivalent to

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

---

**Evaluating** $x + 1 > 2 \ || \ x + 1 < -3$

- Using the precedence rules

  - 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
    \[
    (x >= 0) \land (y > 1)
    \]
    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 \( \text{pieces} / 0 \geq 2 \)
    
    - 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 `(! time > limit), with limit = 60, is evaluated as
  `(!time) > 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 > limit`
    `false`
Correcting the ! Problem

• The intent of the previous expression was most likely the expression

\[( !( time > limit))\]

which evaluates as

\[( ! ( false))\]

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
Simple Loops

- When an action must be repeated, a loop is used
- C++ includes several ways to create loops
- We start with the while-loop

Example:
```c++
while (count_down > 0)
{
    cout << "Hello ";
    count_down -= 1;
}
```

Output: Hello Hello Hello when count_down starts at 3
While Loop Operation

• First, the boolean expression is evaluated
  – If false, the program skips to the line following the while loop
  – If true, the body of the loop is executed
    • During execution, some item from the boolean expression is changed
  – After executing the loop body, the boolean expression is checked again repeating the process until the expression becomes false

• A while loop might not execute at all if the boolean expression is false on the first check

while Loop Syntax

• while (boolean expression is true)
  {
    statements to repeat
  }

  – Semi-colons are used only to end the statements within the loop

• while (boolean expression is true)
  statement to repeat
do-while loop

• A variation of the while loop

• A do-while loop is always executed at least once
  – The body of the loop is first executed
  – The boolean expression is checked after the body has been executed

• Syntax:

```cpp
do
{
    statements to repeat
}
while (boolean_expression);
```
Syntax of the do-while Statement

A Loop Body with Several Statements:

```
do
{
    Statement_1
    Statement_2
    ...
    Statement_Last
} while (Boolean_Expression);
```

A Loop Body with a Single Statement:

```
do
    Statement
while (Boolean_Expression);
```

Do not forget the final semicolon.

A do-while Loop

```cpp
#include <iostream>
using namespace std;

int main()
{
    char ans;
    do
    {
        cout << "Hello!n";
        cout << "Do you want another greeting? (y/n)"n"; // Press y for yes, n for no, and then press return: 
y
        cin >> ans;
    } while (ans == 'y' || ans == 'Y');
    cout << "Good-Bye!n";
    return 0;
}
```

Sample Dialogue

```
Hello
Do you want another greeting?
Press y for yes, n for no, and then press return: y
Hello
Do you want another greeting?
Press y for yes, n for no, and then press return: Y
Hello
Do you want another greeting?
Press y for yes, n for no, and then press return: m
Good-Bye
```
Increment/Decrement

- Unary operators require only one operand
  - + in front of a number such as +5
  - - in front of a number such as -5

- ++ increment operator
  - Adds 1 to the value of a variable
    - `x++;`
    - is equivalent to `x = x + 1;`

- -- decrement operator
  - Subtracts 1 from the value of a variable
    - `x--;`
    - is equivalent to `x = x - 1;`

Sample Program

- Bank charge card balance of $50
- 2% per month interest
- How many months without payments before your balance exceeds $100
- After 1 month: $50 + 2% of $50 = $51
- After 2 months: $51 + 2% of $51 = $52.02
- After 3 months: $52.02 + 2% of $52.02 …
Infinite Loops

- Loops that never stop are infinite loops
- The loop body should contain a line that will eventually cause the boolean expression to become false

Example: Print the odd numbers less than 12

```cpp
x = 1;
while (x != 12)
{
    cout << x << endl;
    x = x + 2;
}
```

- Better to use this comparison: `while (x < 12)`
2.5

Program Style

- A program written with attention to style
  - is easier to read
  - easier to correct
  - easier to change
Program Style - Indenting

- Items considered a group should look like a group
  - Skip lines between logical groups of statements
  - Indent statements within statements

    ```
    if (x == 0)
    statement;
    ```

- Braces {} create groups
  - Indent within braces to make the group clear
  - Braces placed on separate lines are easier to locate

Program Style - Comments

- `//` is the symbol for a single line comment
  - Comments are explanatory notes for the programmer
  - All text on the line following `//` is ignored by the compiler
  - **Example:** `//calculate regular wages
gross_pay = rate * hours;`

- `/*` and `*/` enclose multiple line comments
  - **Example:**
    ```
    /* This is a comment that spans multiple lines without a comment symbol on the middle line
    */
    ```
Program Style - Constants

- Number constants have no mnemonic value
- Number constants used throughout a program are difficult to find and change when needed
- Constants
  - Allow us to name number constants so they have meaning
  - Allow us to change all occurrences simply by changing the value of the constant

Constants

- `const` is the keyword to declare a constant

- Example:

  ```
  const int WINDOW_COUNT = 10;
  ```

  declares a constant named `WINDOW_COUNT`

  - Its value cannot be changed by the program like a variable
  - It is common to name constants with all capitals
// File Name: health.cpp (Your system may require some suffix other than .cpp.)
// Author: Your Name Goes Here.
// Email Address: you@yourmachine.blah.blah
// Assignment Number: 2
// Description: Program to determine if the user is ill.
// Last Changed: September 21, 2004

#include <iostream>
using namespace std;

int main()
{
    double NORMAL = 98.6; // degrees Fahrenheit
    double temperature;
    cout << "Enter your temperature: ";
    cin >> temperature;
    if (temperature > NORMAL)
    {
        cout << "You have a fever.\n";
        cout << "Drink lots of liquids and get to bed.\n";
    }
    else
    {
        cout << "You don't have a fever.\n";
        cout << "Go study.\n";
    }
    return 0;
}

Sample Dialogue

Enter your temperature: 98.6
You don't have a fever.
Go study.