CS 31:  
Introduction To Computer Science I  
Howard A. Stahl

Agenda
• Revisiting Output and Input
• Type Compatibility and Conversion  
• Expressions and Precedence Rules
• Selective Control
• Repetition

Revisiting C++ Output
• cout is connected to the terminal screen
  ```cpp
  cout << expr1 << ... << exprn;
  ```
• `<<` referred to as the insertion operator
• expressions are normally variables or literals
• the identifier `endl` can be used to send a new-line-character and flush the buffer
Revisiting C++ Output

• Examples:
  - `cout << "Hello, World\n";`
  - `cout << "Hello" << "," << " World" << endl;`
  - `cout << "1" << endl;`
  - `cout << 1 << endl;`
  - `cout << 5*3 << endl;`
  - `cout << 1 << 1 << endl;`

Escape Sequences

• Characters Following A Backslash Have A Different Meaning From The Character Themselves

Escape Sequences

<table>
<thead>
<tr>
<th>SEQUENCE</th>
<th>MEANING</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>\n</code></td>
<td>New line</td>
</tr>
<tr>
<td><code>\r</code></td>
<td>Carriage return (Ethernet the cursor at the start of the current line. You are not likely to see this very much.)</td>
</tr>
<tr>
<td><code>\t</code></td>
<td>Horizontal tab ( Advances the cursor to the next tab stop. )</td>
</tr>
<tr>
<td><code>\b</code></td>
<td>Backspace (Backs the alert erase, typically a bell. )</td>
</tr>
<tr>
<td><code>\&quot;</code></td>
<td>Backslash (Allows you to place a backslash in a quoted expression. )</td>
</tr>
</tbody>
</table>
**Escape Sequences**

<table>
<thead>
<tr>
<th>Escape Sequence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\&quot;</td>
<td>Single quote (mostly used to place a single quote inside single quotes.)</td>
</tr>
<tr>
<td>&quot;</td>
<td>Double quote (mostly used to place a double quote inside a quoted string.)</td>
</tr>
</tbody>
</table>

The following are not as commonly used, but we include them for completeness:

<table>
<thead>
<tr>
<th>Escape Sequence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\v</td>
<td>Vertical tab</td>
</tr>
<tr>
<td>\b</td>
<td>Backspace</td>
</tr>
<tr>
<td>\f</td>
<td>Form feed</td>
</tr>
<tr>
<td>?</td>
<td>Question mark</td>
</tr>
</tbody>
</table>

**“Magic Formula”**

- Use the following to output 2 digits after the decimal point when working with cout
  - `cout.setf( ios::fixed );`
  - `cout.setf( ios::showpoint );`
  - `cout.precision( 2 );`

**Revisiting C++ Input**

- `cout` is connected to the terminal screen
  - `cin >> var1 >> ... >> varn ;`
- `>>` referred to as the extraction operator
- Variables are assigned values from standard input
- Values read are separated by whitespace
  - spaces, tabs, CR
- `cin` is greedy
Revisiting C++ Input

- Examples:
  - `cin >> fahrenheit;`
  - `cin >> x_coord >> y_coord;`
- Always better to issue prompt for input
  - `cout << "Enter temperature: \";`
  - `cin >> fahrenheit;`
  - `char symbol1, symbol2;`
  - `cout << "Enter your initials:\";`
  - `cin >> symbol1 >> symbol2;`

Revisiting Character Data

- Character Literals use single quote
  - `'A'  '5'  '?'  '\n'`
- Character variables can be assigned character literal values
  - `char first;`
  - `char last;`
  - `first = 'P';`
  - `last = ' ';`

Revisiting String Data

- String Literals use double quotes
  - "Hello World"
  - "Thank You, Maam"
Revisiting String Data

- C++ has a data type of “string” to store sequences of characters
  - Not a primitive data type - A distinction that will become much more important later on...
  - Must say: `#include <string>`
  - Operator `+` when working on strings will concatenate two strings together
  - `cin >> aString` reads only up to the first whitespace character (tab, space, newline)

I/O Example

```
#include <string>

int main()
{
    string name;
    string dogAge;
    string dogName;
    cout << "How many years old is your dog? " << endl;
    cin >> dogAge;
    cout << "What is your dog's name? " << endl;
    cin >> dogName;
    cout << dogAge << " is approximately equivalent to a " << dogAge / 7 << " year old human."
    return 0;
}
```

I/O Example

```
#include <string>

int main()
{
    string name;
    string dogAge;
    string dogName;
    cout << "How many years old is your dog? " << endl;
    cin >> dogAge;
    cout << "What is your dog's name? " << endl;
    cin >> dogName;
    cout << dogAge << " is approximately equivalent to a 70 year old human."
    return 0;
}
```
Important String Handling Detail

- `cin >>` eats leading whitespace but breaks on whitespace
- `getline( cin, aString )` reads a textual line ending with newline, consuming the newline character itself

- `cin.ignore( 1000, '\n' )` discards up to and including the next \n character or 1000 characters, which ever comes first

- `cin >> actualAge;
- `cin.ignore( INT_MAX, '\n' );
- `getline( cin, dogName );
- `getline( cin, dogBreed )`
Important String Handling Detail

• `cin >> actualAge;`
• `cin.ignore( INT_MAX, \n`);
• `getline( cin, dogName );`
• `getline( cin, dogBreed );`

If You Use `INT_MAX`
Be Sure To:
`#include <climits>`

Constants

• It’s a good idea to name values to prevent “magic” values showing up in your code
• Use `const` declaration to state that value cannot change after assignment
• Examples:
  `const double PI = 3.14159;`
  `const int LIMIT = 15;`

Type Compatibility

• Generally, you should not try storing values of one type in a variable of a different type
  – Type Mismatch Error
• Storing a `double` in `int` leads to truncation
• Storing an `int` in a `double` is OK
  – best to convert
Type Compatibility

• You can coerce types from one to another by saying:

```cpp
double value( 12.510104 );
int i=static_cast<int>( value );
```

Type Compatibility

• Need to be careful if you mix types and values on assignment statements or arithmetic expressions

• When working with
  
  \[ A \text{ operand } B \]

  where operand may be +, -, *, /, or %

  – if \( A \) or \( B \) is \text{double}, the result will be \text{double}

Type Compatibility

• Examples:
  
  – 3 + 4.4 =
  – 2.2 * 3 =
  – 2.2 * 3.0 =
  – 2 * 3 =
  – 4.5 * 2 =
  – 9 * 2 =
Division

- When working with $\frac{A}{B}$
- If either operand is real, then the other will be converted to a real and the result will be real
- If both are int, then integer division occurs and the result will be an int
  - modulus operator $\%$ yields the remainder

Division

- Examples:
  - $9 / 4 =$
  - $9.0 / 4 =$
  - $9 \% 4 =$
  - $11 / 4 =$
  - $11 \% 4 =$
  - $-11 \% 4 =$
  - TRICK QUESTION: `var_a * (1 / 4) =`

Precedence Rules

- Operators in an expression are evaluated according to precedence rules
  - $(())$
  - $*,/,$%
  - $+, -, -=, +=, \cdot, /=, -=, %=\$
- Precedence described in Appendix 2, page 917
### Precedence Rules

#### Display 2.3 Precedence of Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>( )</td>
<td>Highest precedence (innermost)</td>
</tr>
<tr>
<td>.</td>
<td>Dot operator</td>
</tr>
<tr>
<td>,</td>
<td>Array index</td>
</tr>
<tr>
<td>[ ]</td>
<td>Array indexing/lookup</td>
</tr>
<tr>
<td>:=</td>
<td>Pre/Post increment operator (postfix before the variable)</td>
</tr>
<tr>
<td>=</td>
<td>Pre/Post decrement operator (prefix before the variable)</td>
</tr>
<tr>
<td>+</td>
<td>Addition</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
</tr>
<tr>
<td>%</td>
<td>Remainder (modulus)</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>Insertion operator (console output)</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>Extraction operator (console input)</td>
</tr>
</tbody>
</table>

#### Display 2.3.1 Precedence of Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>Less than</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal to</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal to</td>
</tr>
<tr>
<td>==</td>
<td>Equal</td>
</tr>
<tr>
<td>!=</td>
<td>Not equal</td>
</tr>
<tr>
<td>&amp;</td>
<td>and</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Precedence Rules

- Assignment
- Add and assign
- Subtract and assign
- Multiply and assign
- Divide and assign
- Modulo and assign

? : Conditional operator

Flow of Control

- Like a cook following recipe instructions, computers execute statements one after another
- Certain statements alter this flow of control
  – if
  – if-else
  – while
  – do-while

“Shorthand” Operators

- Calculate And Assign

<table>
<thead>
<tr>
<th>EXAMPLE</th>
<th>EQUIVALENT TO</th>
</tr>
</thead>
<tbody>
<tr>
<td>count -= 2;</td>
<td>count = count + 2;</td>
</tr>
<tr>
<td>total -= discount;</td>
<td>total = total - discount;</td>
</tr>
<tr>
<td>bonus -= 2;</td>
<td>bonus = bonus + 2;</td>
</tr>
<tr>
<td>time /= rushFactor;</td>
<td>time = time/rushFactor;</td>
</tr>
<tr>
<td>change %= 100;</td>
<td>change = change % 100;</td>
</tr>
<tr>
<td>amount -= cnt1 + cnt2;</td>
<td>amount = amount * (cnt1 + cnt2);</td>
</tr>
</tbody>
</table>
Selective Control Flow in C++

• Programs often choose between different instructions in a variety of situations
  – sometimes, code must be skipped because it does not apply in the current situation
  – other times, one of several code blocks must be chosen to be executed based on the current situation

The if Statement

• Guarded Action
  if ( x < y )
  {
    cout<<"x < y";
  }
The if Statement

• Guarded Action

```
if ( x < y )
{
    cout<<"x < y";
}
```

Logical Test
The if Statement

- Guarded Action
  
  ```cpp
  if (x < y)
  {
    cout << "x < y";
  }
  ```

Comparison Operators

- Testing Ordering
  - `<`, `<=`, `>`, `>=`

- Testing Equality
  - `==`, `!=`

<table>
<thead>
<tr>
<th>OP</th>
<th>ORDER</th>
<th>C++ OPERATOR</th>
<th>C++ NAME</th>
<th>JAVA OPERATOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>Equal</td>
<td>x = 7 == 2y</td>
<td>x &lt; y</td>
<td></td>
</tr>
<tr>
<td>!=</td>
<td>Not equal to</td>
<td>x = &quot;a&quot;</td>
<td>x != &quot;a&quot;</td>
<td></td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal to</td>
<td>x = 3</td>
<td>x &lt;= 3</td>
<td></td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
<td>x = 10</td>
<td>x &gt; 10</td>
<td></td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal to</td>
<td></td>
<td>x &gt;= 10</td>
<td></td>
</tr>
<tr>
<td>=</td>
<td>Equal to</td>
<td>x = 20</td>
<td>x = 20</td>
<td></td>
</tr>
</tbody>
</table>

Common Mistake

- Assignment (`=`) is different from Equality (`==`)
  
  ```cpp
  if (salary = 100000)
  {
    cout << "You're fired!";
  }
  ```

- Equality is always dangerous when working with real operands
More Complex Expressions

• Examples:
  - if (rate * balance > 1000)
  - if (a * b != c + d * e)
  - if (a / b > c)

• Never hurts to add parenthesis to make your intentions clear.
• Arithmetic operators have higher precedence than relational operators.
  - 24.00000001 != 24

Logical Operators

• && means AND, || means OR, ! means NOT
• Examples:
  - true and false =
  - false and true =
  - true or false =
  - false or true =
  - not true =
  - not false =

<table>
<thead>
<tr>
<th>Operator</th>
<th>Truth Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;&amp; (AND)</td>
<td>True, True</td>
</tr>
<tr>
<td></td>
<td>False, False</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>False, True</td>
</tr>
<tr>
<td>! (NOT)</td>
<td>True, False</td>
</tr>
<tr>
<td></td>
<td>False, True</td>
</tr>
</tbody>
</table>
Logical Operators

- Logical Operators connect expressions
- Examples:
  
  ```
  if ((0 <= x) && (x > 3))
  if ((y != 1) && (x/y > 4))
  ```
- C++ uses short-circuit evaluation
  - The evaluation of condition stops because the condition turns false (in case of &&) or true (in case of ||)

Precedence Rules

- Parentheses
- Unary Operators: +, -, !
- Arithmetic Operators: *, / then +, -, then %
- Comparison Operators: <, <=, >, >=, ==, !=
  then && then ||
- See Appendix 2 for full set of rules

Time For Our Next Demo!

- Selection.cpp
Summarizing Our Second Demo!

• Proper Indentation Helps Express Your Intentions
  – But Remember, The Computer Cares Little For Whitespace....

The \texttt{if-else} Statement

• Alternative Action
  \begin{verbatim}
  if ( x < y )
  { x++;
  }
  else
  { y++;
  }
  \end{verbatim}
The `if-else` Statement

- **Logical Test**
  
  if (x < y)
  {
    x++;
  }
  else
  {
    y++;
  }

• **Alternative Action**

- **Logical Test**

  if (x < y)
  {
    x++;
  }
  else
  {
    y++;
  }

- **Any Block Of C++ Statements**
The if-else Statement

- Alternative Action

```cpp
if (x < y)
{
    x++;  // true branch
}
else
{
    y++;  // false branch
}
```

Multiway if-else Statement

**Syntax**
```
if (Boolean_Expression_1)
{
    Statement_1;
}
else if (Boolean_Expression_2)
{
    Statement_2;
}
else if (Boolean_Expression_n)
{
    Statement_n;
}
else // no more alternative actions
```

**Example**
```
if (Temperature < -10) && (day == SUNDAY)
    cout << "Stay home."
else if (Temperature < -10) && (day != SUNDAY)
    cout << "Stay home, but call work.");
else if (Temperature > 80 // and temperature >= -10
    cout << "Drink water."
else if (Temperature > 0)
    cout << "Work hard and play hard."
```

The Boolean expressions are checked in order until the first true Boolean expression is encountered, and then the corresponding statement is executed. If none of the Boolean expressions is true, then the `Statement_if_all.Other_Actions` is executed.
Nested Conditional Statements

- Selection Statements can be used in combination
- Just be sure that the else clause is not dangling...
  if (precipitating)
    if (temperature < 32)
      cout << "It's snowing";
  else // HMMM...
    cout << "It's raining";

Time For Our Next Demo!

- Nesting.cpp

(See Handout For Example 3)

Summarizing Our Third Demo!

- Nested Conditionals Make For Complex Scenarios
- Use Parentheses To Prevent A Dangling else
- Remember Only One Guarded Action Or Alternative Is Chosen
Repetitive Control Flow in C++

- Programs often must repeat different instructions in a variety of situations
  - sometimes, code must be repeated a determinate number of times
  - other times, code must be repeated an indeterminate number of times

The while Statement

- Indeterminate Loop
  - Repeat While A Condition Is True

```cpp
while ( logical-expression ) {
  ...block of statements...
}
```

The while Statement

- Indeterminate Loop

```cpp
while (x < y) {
  cout << "x<y\n";
  x++;
}
```
The while Statement

- Indeterminate Loop
  while (x < y) {
    cout << "x<y\n";
    x++;
  }

Logical Test

true
false
The while Statement

- Indeterminate Loop
  ```cpp
  while (x < y) {
    cout << "x<y\n";
    x++;
  }
  ```

  Logical Test
The **while** Statement

Syntax for while and do-while Statements

A while STATEMENT WITH A SINGLE STATEMENT BODY

```cpp
while (Boolean_Expression)
    Statement
```

A while STATEMENT WITH A MULTISTATEMENT BODY

```cpp
while (Boolean_Expression)
    { Statement1; Statement2; ... Statement_last }
```

Copyright © 2016 Pearson Inc.

---

The **do...while** Statement

- **Indeterminate Loop**
  - Repeat While A Condition Is True

```cpp
do {
    ...block of statements...
} while (logical-expression);
```

---

The **do...while** Statement

- **Indeterminate Loop**

```cpp
do {
    cout << "x<y\n";
    x++;
} while (x < y);
```
The do...while Statement

- Indeterminate Loop
  ```cpp
do {
    cout << "x<y\n";
    x++;
} while (x < y);
```

• Indeterminate Loop
  ```cpp
do {
    cout << "x<y\n";
    x++;
} while (x < y);
```

• Indeterminate Loop
  ```cpp
do {
    cout << "x<y\n";
    x++;
} while (x < y);
```

• Indeterminate Loop
  ```cpp
do {
    cout << "x<y\n";
    x++;
} while (x < y);
```
The do...while Statement

- Indeterminate Loop
  do {
    cout << "x<y\n";
    x++;
  } while (x < y);

• Indeterminate Loop
  do {
    cout << "x<y\n";
    x++;
  } while (x < y);

• Indeterminate Loop
  do {
    cout << "x<y\n";
    x++;
  } while (x < y);

• Indeterminate Loop
  do {
    cout << "x<y\n";
    x++;
  } while (x < y);

• Indeterminate Loop
  do {
    cout << "x<y\n";
    x++;
  } while (x < y);
The do...while Statement

- Indeterminate Loop

```cpp
do {
    cout << "x<y\n";
    x++;
} while (x < y);
```

The do...while Statement

- A do-while STATEMENT WITH A SINGLE-STATEMENT BODY
- A do-while STATEMENT WITH A MULTISTATEMENT BODY

Time For Our Next Demo!

- Loops.cpp

(See Handout For Example 3)
Summarizing Our Third Demo!

- Typically, one of the loop forms fits your problem better than the other
- However, any loop written in one form can be re-written in the other

while versus do...while

- while loop may never execute
- do...while loop will always execute at least once

When To Use Loops

- Whenever you have a task to do repeatedly
  - "As long as some condition is true, do some action..."
  - "Do some action until some condition is no longer true..."
- Sometime, looping is harder to recognize
  - For a given value in cents (0 to 99), calculate how many quarters, dimes, nickels and pennies are required to represent that value
How To Use Loops

• Identify the terminating condition
  – how will the loop stop?
• Identify the initial condition
  – what is true before the loop ever executes?
• How is progress made toward the
  terminating condition
  – something must guarantee progress toward the
  terminating condition
  – without progress, you will have an infinite loop

Summary

• Revisiting Output and Input
• Type Compatibility and Conversion
• Expressions and Precedence Rules
• Selective Control
• Repetition