CSCI 1061U
Programming Workshop 2

C++ Basics
Learning Objectives

• Introduction to C++
  • Origins, Object-Oriented Programming, Terms

• Variables, Expressions, and Assignment Statements

• Console Input/Output

• Program Style

• Libraries and Namespaces
#include <iostream>

using namespace std;

int main() {
    int numberOfLanguages;

    cout << "Hello reader.\n" << "Welcome to C++.\n";

    cout << "How many programming languages have you used? ";
    cin >> numberOfLanguages;

    if (numberOfLanguages < 1)
        cout << "Read the preface. You may prefer\n" << "a more elementary book by the same author.\n";
    else
        cout << "Enjoy the book.\n";

    return 0;
}
C++ Identifiers and Variables

- Identifiers (variable and function names) are case sensitive
- Reserved words cannot be used as an identifier
  - E.g., it is not possible to name a variable int
- Variables must be declared before these can be used

Slide Credit: Copyright © 2016 Pearson Inc. All rights reserved.
## Data Types

<table>
<thead>
<tr>
<th>TYPE NAME</th>
<th>MEMORY USED</th>
<th>SIZE RANGE</th>
<th>PRECISION</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>short</strong> (also called short int)</td>
<td>2 bytes</td>
<td>-32,768 to 32,767</td>
<td>Not applicable</td>
</tr>
<tr>
<td><strong>int</strong></td>
<td>4 bytes</td>
<td>-2,147,483,648 to 2,147,483,647</td>
<td>Not applicable</td>
</tr>
<tr>
<td><strong>long</strong> (also called long int)</td>
<td>4 bytes</td>
<td>-2,147,483,648 to 2,147,483,647</td>
<td>Not applicable</td>
</tr>
<tr>
<td><strong>float</strong></td>
<td>4 bytes</td>
<td>approximately $10^{-38}$ to $10^{38}$</td>
<td>7 digits</td>
</tr>
<tr>
<td><strong>double</strong></td>
<td>8 bytes</td>
<td>approximately $10^{-308}$ to $10^{308}$</td>
<td>15 digits</td>
</tr>
</tbody>
</table>
## Data Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Size</th>
<th>Range</th>
<th>Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>long double</code></td>
<td>8 bytes</td>
<td>approximately $10^{-4932}$ to $10^{4932}$</td>
<td>19 digits</td>
</tr>
<tr>
<td><code>char</code></td>
<td>1 byte</td>
<td>All ASCII characters (Can also be used as an integer type, although we do not recommend doing so.)</td>
<td>Not applicable</td>
</tr>
<tr>
<td><code>bool</code></td>
<td>1 byte</td>
<td><code>true</code>, <code>false</code></td>
<td>Not applicable</td>
</tr>
</tbody>
</table>

The values listed here are only sample values to give you a general idea of how the types differ. The values for any of these entries may be different on your system. *Precision* refers to the number of meaningful digits, including digits in front of the decimal point. The ranges for the types `float`, `double`, and `long double` are the ranges for positive numbers. Negative numbers have a similar range, but with a negative sign in front of each number.
C++11 Fixed Width Integer Types

- Avoids problem of variable integer sizes for different CPUs

<table>
<thead>
<tr>
<th>TYPE NAME</th>
<th>MEMORY USED</th>
<th>SIZE RANGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>int8_t</td>
<td>1 byte</td>
<td>-128 to 127</td>
</tr>
<tr>
<td>uint8_t</td>
<td>1 byte</td>
<td>0 to 255</td>
</tr>
<tr>
<td>int16_t</td>
<td>2 bytes</td>
<td>-32,768 to 32,767</td>
</tr>
<tr>
<td>uint16_t</td>
<td>2 bytes</td>
<td>0 to 65,535</td>
</tr>
<tr>
<td>int32_t</td>
<td>4 bytes</td>
<td>-2,147,483,648 to 2,147,483,647</td>
</tr>
<tr>
<td>uint32_t</td>
<td>4 bytes</td>
<td>0 to 4,294,967,295</td>
</tr>
<tr>
<td>int64_t</td>
<td>8 bytes</td>
<td>-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807</td>
</tr>
<tr>
<td>uint64_t</td>
<td>8 bytes</td>
<td>0 to 18,446,744,073,709,551,615</td>
</tr>
<tr>
<td>long long</td>
<td>At least 8 bytes</td>
<td></td>
</tr>
</tbody>
</table>
New C++11 Types

• auto
  • Deduces the type of the variable based on the expression on the right side of the assignment statement
    ```
    auto x = expression;
    ```
  • More useful later when we have verbose types

• decltype
  • Determines the type of the expression. In the example below, x*3.5 is a double so y is declared as a double.
    ```
    decltype(x*3.5) y;
    ```
Data Assignment

• Assignment operator (=) is used to assign value to a variable

• Assignment can take place during or after declaration
Data Assignment

```c++
var4 = var3 * var1 * var2 * 2;
```

**L-value**
- Must be a variable

**R-value**
- Any valid expression
Assigning Data: Shorthand Notations

<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>
Data Assignment Rules

• Generally speaking type mismatches are not allowed
  • Cannot place value of one type into variable of another type

• Special case – implicit or automatic type conversions allow us to place value of one type into variable of another type
Literal Data

• Cannot change values during execution
• Called "literals" because you "literally typed" them in your program!
Escape Sequences

• "Extend" character set
• Backslash (\) preceding a character
  • Instructs compiler: a special "escape character" is coming
• Following character treated as "escape sequence char"

<table>
<thead>
<tr>
<th>SEQUENCE</th>
<th>MEANING</th>
</tr>
</thead>
<tbody>
<tr>
<td>\n</td>
<td>New line</td>
</tr>
<tr>
<td>\r</td>
<td>Carriage return (Positions the cursor at the start of the current line. You are not likely to use this very much.)</td>
</tr>
<tr>
<td>\t</td>
<td>(Horizontal) Tab (Advances the cursor to the next tab stop.)</td>
</tr>
<tr>
<td>\a</td>
<td>Alert (Sounds the alert noise, typically a bell.)</td>
</tr>
<tr>
<td>&quot;</td>
<td>Backslash (Allows you to place a backslash in a quoted expression.)</td>
</tr>
<tr>
<td>'</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>SEQUENCE</th>
<th>MEANING</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>
Raw String Literals

• Introduced with C++11
• Avoids escape sequences by literally interpreting everything in parens

    string s = R“(\t\t\n)”;

• The variable s is set to the exact string “\t\t\n”
• Useful for filenames with \ in the filepath
Constants

• Literal constants are "OK", but provide little meaning
• Use named constants instead
  • Meaningful name to represent data
  • Called a "declared constant" or "named constant"
  • Now use it’s name wherever needed in program
  • Added benefit: changes to value result in one fix
Arithmetic Precision

• Precision of Calculations
• VERY important consideration!
• Expressions in C++ might not evaluate as you’d "expect"!
• "Highest-order operand" determines type of arithmetic "precision" performed
• Common pitfall!
Arithmetic Precision Examples

• 17 / 5 evaluates to 3 in C++!
  • Both operands are integers
  • Integer division is performed!

• 17.0 / 5 equals 3.4 in C++!
  • Highest-order operand is "double type"
  • Double "precision" division is performed!

• int intVar1 =1, intVar2=2;
  intVar1 / intVar2;
  • Performs integer division!
  • Result: 0!
Individual Arithmetic Precision

• Calculations done "one-by-one"
  • $1 / 2 / 3.0 / 4$ performs 3 separate divisions.
    • First $1 / 2$ equals 0
    • Then $0 / 3.0$ equals 0.0
    • Then $0.0 / 4$ equals 0.0!

• So not necessarily sufficient to change just "one operand" in a large expression

• Must keep in mind all individual calculations that will be performed during evaluation!
Type Casting

• Can add ".0" to literals to force precision arithmetic, but what about variables?
Type Casting

- Implicit—also called "Automatic"
  - Done FOR you, automatically
    17 / 5.5
    This expression causes an "implicit type cast" to take place, casting the 17 → 17.0

- Explicit type conversion
  - Programmer specifies conversion with cast operator
    (double)17 / 5.5
    Same expression as above, using explicit cast
    (double)myInt / myDouble
    More typical use; cast operator on variable
Shorthand Operators

- Post increment and decrement
  
  ```
  i++; // i=i+1;
  k--; // k=k-1;
  ```

- Pre increment and decrement
  
  ```
  ++i; // i=i+1;
  --k; // k=k-1;
  ```
Console Input/Output

- **cin** – used to read from console
- **cout** – used to write to console
- **cerr** – used to write to console, typically used for error messages

Must include these two lines. `cin`, `cout` and `cerr` are defined in std namespace and are found in iostream header file.
Console Input/Output

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

int main()
{
    int n;
    cout << "Enter a number a number less than 10: ";
    cin >> n;
    if (n < 10) {
        cout << "You entered " << n << endl;
    } else {
        cerr << "Invalid entry." << endl;
    }
    return 0;
}
```

```cpp
#include <iostream>

int main()
{
    int n;
    std::cout << "Enter a number a number less than 10: ";
    std::cin >> n;
    if (n < 10) {
        std::cout << "You entered " << n << std::endl;
    } else {
        std::cerr << "Invalid entry." << std::endl;
    }
    return 0;
}
```

Spot the differences
Console Output

• Any data can be outputted to display screen
  • Variables
  • Constants
  • Literals
  • Expressions (which can include all of above)
• Cascading (multiple values in one cout) is allowed
Console Output

- Any data can be outputted to display screen
  - Variables
  - Constants
  - Literals
  - Expressions (which can include all of above)
- Cascading (multiple values in one cout) is allowed

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

int main()
{
    float g = 9.8;
    cout << "gravity " << "of " << "earth is " << g << " m/s^2 \n";
    cout << 10 + 23 + 192 << " is allowed " << endl;
    return 0;
}
```
String type

• C++ has a data type of “string” to store sequences of characters
  • Not a primitive data type (more on that later)
• Use cin to read strings from console
  • Up to the first “space”

String defined in std namespace and is found in string header file.
Formatting Numbers

• We can explicitly tell C++ how to output numbers in our programs
  • How many decimal places, etc.
Input Using cin

- Extraction operator ">>" (extraction operator) points toward where the data goes
- Must input "to a variable”, literals are not allowed.
- `cin >> num;
  - Waits on-screen for keyboard entry
  - Value entered at keyboard is "assigned" to num

```
int x;
cin >> x;    // Ok
```
```
cin >> "9";  // Error
```
Prompting for Input: cin and cout

• Always "prompt" user for input

```cpp
cout << "Enter number of dragons: ";
cin >> numOfDragons;
```

• Note no "\n" in cout. Prompt "waits" on same line for keyboard input as follows (underscore below denotes where keyboard entry is made):

```
Enter number of dragons: _____
```
Program Style

• Bottom-line: Make programs easy to read and modify

• Comments, two methods:
  • // Two slashes indicate entire line is to be ignored
  • /* Delimiters indicates everything between is ignored */
  • Both methods commonly used

• Identifier naming
  • ALL_CAPS for constants
  • lowerToUpper or parta_partb for variables
  • Most important: MEANINGFUL NAMES!
Libraries

- C++ Standard Libraries
  - `#include <Library_Name>`
    - Directive to "add" contents of library file to your program
    - Called "preprocessor directive"
      - Executes before compiler, and simply "copies" library file into your program file
- C++ has many libraries
  - Input/output, math, strings, etc.
Namespaces

• Namespaces defined:
  – Collection of name definitions

• For now: interested in namespace "std"
  – Has all standard library definitions we need

Includes entire standard library of name definitions
#include <iostream>
using namespace std;

Can specify just the objects we want
#include <iostream>
using std::cout;
using std::cin;
Summary 1

• C++ is case-sensitive

• Use meaningful names
  • For variables and constants

• Variables must be declared before use
  • Should also be initialized

• Use care in numeric manipulation
  • Precision, parentheses, order of operations

• #include C++ libraries as needed
Summary 2

- Object `cout`
  - Used for console output

- Object `cin`
  - Used for console input

- Object `cerr`
  - Used for error messages

- Use comments to aid understanding of your program
  - Do not overcomment
CSCI 1061U
Programming Workshop 2

Flow Control
Learning Objectives

• Boolean Expressions
  • Building, Evaluating & Precedence Rules

• Branching Mechanisms
  • if-else
  • switch
  • Nesting if-else

• Loops
  • While, do-while, for
  • Nesting loops

• Introduction to File Input
Flow control

• Branching
  • if else elseif switch

• Loops
  • for while do-while
# 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>≤</td>
<td>Less than or equal to</td>
<td>&lt;=</td>
<td>time &lt;= limit</td>
<td>time ≤ limit</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>≥</td>
<td>Greater than or equal to</td>
<td>&gt;=</td>
<td>age &gt;= 21</td>
<td>age ≥ 21</td>
</tr>
</tbody>
</table>
Boolean Expressions

• Use `&&` for Boolean AND operator
• Use `||` for Boolean OR operator
• Use data type `bool` to store Boolean values
• Boolean expressions return either true or false
  • true, false are predefined library consts
Branching: if-else

```
if (<Boolean expression>)
  <statement>;
else
  <statement>;
```

Only one statement allowed in each block.

```
if (<Boolean expression>) {
  <statement1>;
  <statement2>;
}
else {
  <statement1>;
  <statement2>;
}
```

Use `{}` block when using multiple statements.
Branching: if-else

```cpp
if (<Boolean expression>)
    <statement>;
else
    <statement>;

if (<Boolean expression>) {
    <statement1>;
    ...
    <statementN>;
} else {
    <statement1>;
    ...
    <statementN>;
}
```

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

int main()
{
    int n;
    cout << "Enter a number between 0 and 10: ";
    cin >> n;
    if (n < 0 || n > 10) { // Using {} block, since more than one statement
        cout << "Invalid entry." << endl;
        cout << "Sorry cannot proceed." << endl;
    }
    else { // Not using {} block, since only one statement
        cout << "You entered " << n << endl;
        return 0;
    }
}
```
Branching: if-else if-else

```cpp
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
```

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

int main()
{
    int n;
    cout << "Enter a number between 0 and 10: ";
    cin >> n;
    if (n < 0 || n > 10) { // Using a {} block
        cout << "Invalid entry." << endl;
    } else if (n % 2 == 0) {
        cout << n << " is even." << endl;
    } else // Not using {} block here
        cout << n << " is odd." << endl;
    return 0;
}
```
Branching: if

else block is optional.

```cpp
if (<Boolean expression>)
  <statement>;
```
Nested Branch Statements

• It is possible to nest branching statements
Branching: switch

- A statement for controlling multiple branches
- Can do the same thing with if statements but sometimes switch is more convenient

```
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
}
```

You need not place a `break` statement in each case. If you omit a `break`, that case continues until a `break` (or the end of the `switch` statement) is reached.
Branching: switch example

```cpp
int vehicleClass;
double toll;
cout << "Enter vehicle class: ";
cin >> vehicleClass;

switch (vehicleClass)
{
    case 1:
        cout << "Passenger car."
        toll = 0.50;
        break;
    case 2:
        cout << "Bus."
        toll = 1.50;
        break;
    case 3:
        cout << "Truck."
        toll = 2.00;
        break;
    default:
        cout << "Unknown vehicle class!"

    // If you forget this break, then passenger cars will pay $1.50.
}
```
Branching: switch combining cases

- Execution "falls thru" until break
Conditional Operator or Ternary Operator

• Conditional assignment
• Shorthand if-else syntax

```
if (n1 > n2)
    max = n1;
else
    max = n2;
```

```
max = (n1 > n2) ? n1 : n2;
```

Both of these are equivalent
Loops

• 3 Types of loops in C++
  • while
    • Most flexible
    • No "restrictions"
  • do-while
    • Least flexible
    • Always executes loop body at least once
  • for
    • Natural "counting" loop
while Loops Syntax

A while STATEMENT WITH A SINGLE STATEMENT BODY

```
while (Boolean_Expression)
    Statement
```

A while STATEMENT WITH A MULTISTATEMENT BODY

```
while (Boolean_Expression)
{
    Statement_1
    Statement_2
    .
    .
    .
    Statement_Last
}
```
while Loop Example

count = 0; // Initialization
while (count < 3) // Loop Condition
{
    cout << "Hi "; // Loop Body
    count++; // Update expression
}

Slide Credit: Copyright © 2016 Pearson Inc. All rights reserved.
do-while Loop Syntax

A do–while STATEMENT WITH A SINGLE-STATEMENT BODY

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

A do–while STATEMENT WITH A MULTISTATEMENT BODY

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

Do not forget the final semicolon.
do-while Loop Example

count = 0;              // Initialization
do
{
    cout << "Hi ";      // Loop Body
    count++;            // Update expression
} while (count < 3);  // Loop Condition
while vs. do-while

• What is the difference between while and do-while?
Comma Operator

• Evaluate list of expressions, returning value of the last expression

• Most often used in a for-loop

• Example:
  first = (first = 2, second = first + 1);
  • first gets assigned the value 3
  • second gets assigned the value 3

• No guarantee what order expressions will be evaluated.
for Loop Syntax

• A natural "counting" loop

```
for (<Init_Action>; <Bool_Exp>; <Update_Action>)
    Single_Body_Statement
```

```
for (<Init_Action>; <Bool_Exp>; <Update_Action>) {
    Multiple_Body_Statements
}
```
for Loop Example

• for (count=0; count<3; count++)
  {
    cout << "Hi "; // Loop Body
  }

• How many times does loop body execute?

• Initialization, loop condition and update all "built into" the for-loop structure!

• A natural "counting" loop
Loop Pitfalls: Misplaced ;

- Watch the misplaced ; (semicolon)

```c
while (response != 0);
{
    cout << "Enter val: ";
    cin >> response;
}
```

- Result here: INFINITE LOOP!
Loop Pitfalls: Infinite Loops

• Loop condition must evaluate to false at some iteration through loop, otherwise the loop will run *forever*

• Infinite loops can be desirable
  • e.g., "Embedded Systems"

An infinite loop

```cpp
while (1)
{
    cout << "Hello ";
}
```
The break and continue Statements

• break;
  • Forces loop to exit immediately.

• continue;
  • Skips rest of loop body

• Use these statements with caution to break the natural control flow of a loop
Nested Loops

• Any valid C++ statements can be inside body of loop
• This includes additional loop statements, resulting in "nested loops"

```c++
for (outer=0; outer<5; outer++)
    for (inner=7; inner>2; inner--)
        cout << outer << inner;
```
Basic File IO – Reading from a text file

• Add at the top
  ```cpp
  #include <fstream>
  using namespace std;
  ```

• You can then declare an input stream just as you would declare any other variable.
  ```cpp
  ifstream inputStream;
  ```

• Next you must connect the inputStream variable to a text file on the disk.
  ```cpp
  inputStream.open("filename.txt");
  ```

• The “filename.txt” is the pathname to a text file or a file in the current directory
**Reading from a Text File**

- Use
  ```cpp
  inputStream >> var;
  ```
- The result is the same as using `cin >> var` except the input is coming from the text file and not the keyboard.
- When done with the file close it with
  ```cpp
  inputStream.close();
  ```
File Input Example

Display 2.10  Sample Text File, player.txt, to Store a Player’s High Score and Name

100510
Gordon Freeman

Display 2.11  Program to Read the Text File in Display 2.10

```cpp
#include <iostream>
#include <fstream>
#include <string>

using namespace std;

int main() {
    string firstName, lastName;
    int score;
    fstream inputStream;

    inputStream.open("player.txt");
    inputStream >> score;
    inputStream >> firstName >> lastName;

    cout << "Name: " << firstName << " " << lastName << endl;
    cout << "Score: " << score << endl;
    inputStream.close();

    return 0;
}
```

Sample Dialogue

Name: Gordon Freeman
Score: 100510
# Precedence of Operators (1 of 4)

## Display 2.3 Precedence of Operators

<table>
<thead>
<tr>
<th>::</th>
<th>Scope resolution operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>Dot operator</td>
</tr>
<tr>
<td>-&gt;</td>
<td>Member selection</td>
</tr>
<tr>
<td>[]</td>
<td>Array indexing</td>
</tr>
<tr>
<td>( )</td>
<td>Function call</td>
</tr>
<tr>
<td>++</td>
<td>Postfix increment operator (placed after the variable)</td>
</tr>
<tr>
<td>--</td>
<td>Postfix decrement operator (placed after the variable)</td>
</tr>
<tr>
<td>++</td>
<td>Prefix increment operator (placed before the variable)</td>
</tr>
<tr>
<td>--</td>
<td>Prefix decrement operator (placed before the variable)</td>
</tr>
<tr>
<td>!</td>
<td>Not</td>
</tr>
<tr>
<td>–</td>
<td>Unary minus</td>
</tr>
<tr>
<td>+</td>
<td>Unary plus</td>
</tr>
<tr>
<td>*</td>
<td>Dereference</td>
</tr>
<tr>
<td>&amp;</td>
<td>Address of</td>
</tr>
<tr>
<td>new</td>
<td>Create (allocate memory)</td>
</tr>
<tr>
<td>delete</td>
<td>Destroy (deallocate)</td>
</tr>
<tr>
<td>delete[]</td>
<td>Destroy array (deallocate)</td>
</tr>
<tr>
<td>sizeof</td>
<td>Size of object</td>
</tr>
<tr>
<td>( )</td>
<td>Type cast</td>
</tr>
</tbody>
</table>

Highest precedence (done first)
## Precedence of Operators (2 of 4)

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>Multiply</td>
</tr>
<tr>
<td>/</td>
<td>Divide</td>
</tr>
<tr>
<td>%</td>
<td>Remainder (modulo)</td>
</tr>
<tr>
<td>+</td>
<td>Addition</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction</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>

* Lower precedence (done later)"
Precedence of Operators (3 of 4)

Display 2.3  Precedence of Operators

All operators in part 2 are of lower precedence than those in part 1.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</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;&amp;</td>
<td>And</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Precedence of Operators (4 of 4)

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>=</code></td>
<td>Assignment</td>
</tr>
<tr>
<td><code>+=</code></td>
<td>Add and assign</td>
</tr>
<tr>
<td><code>-=</code></td>
<td>Subtract and assign</td>
</tr>
<tr>
<td><code>*=</code></td>
<td>Multiply and assign</td>
</tr>
<tr>
<td><code>/=</code></td>
<td>Divide and assign</td>
</tr>
<tr>
<td><code>%=</code></td>
<td>Modulo and assign</td>
</tr>
<tr>
<td><code>?:</code></td>
<td>Conditional operator</td>
</tr>
<tr>
<td><code>throw</code></td>
<td>Throw an exception</td>
</tr>
<tr>
<td><code>,</code></td>
<td>Comma operator</td>
</tr>
</tbody>
</table>

*Lowest precedence (done last)*
Precedence Examples

• Arithmetic before logical
  • \( x + 1 > 2 \) || \( x + 1 < -3 \) means:
    • \( (x + 1) > 2 \) || \( (x + 1) < -3 \)

• Short-circuit evaluation
  • \( (x \geq 0) \) && \( (y > 1) \)
  • Be careful with increment operators!
    • \( (x > 1) \) && \( (y++) \)

• Integers as boolean values
  • All non-zero values \( \rightarrow \) true
  • Zero value \( \rightarrow \) false
Strong Enum

• C++11 introduces **strong enums** or **enum classes**
• Does not act like an integer

```cpp
enum class Days { Sun, Mon, Tue, Wed, Thu, Fri, Sat };
enum class Weather { Rain, Sun };
Days d = Days::Tue;
Weather w = Weather::Sun;

Illegal:  if (d == 0)
Legal:    if (d == Days::Wed)
```
Summary 1

• Boolean expressions
  • Similar to arithmetic → results in true or false

• C++ branching statements
  • if-else, switch
    • switch statement great for menus

• C++ loop statements
  • while
  • do-while
  • for
Summary 2

• do-while loops
  • Always execute their loop body at least once

• for-loop
  • A natural "counting" loop

• Loops can be exited early
  • break statement
  • continue statement
  • Usage restricted for style purposes

• Reading from a text file is similar to reading from cin
Reading

• Ch. 1–2