Welcome to Lesson 10!

Announcements

- Midterm after a short lesson and the break
- Don't Forget Assignment 9 due Tuesday!
- Don't Forget Lab 5 due Friday

Numbers, Operators and Precision Cont'd

Assignment Operators

- As we discussed before, we assign values to variables using an equal (=) sign

```
int sum = 5;
```

- However, the equal sign is really an assignment operator and does not denote equality
- Thus, unlike math, we can have the same variable on both sides of an equals sign:

```
int sum = 5;  // initialize sum to 25
sum = sum + 10;  // add to sum
```

- Note that the value of the variable is changed in the second line
- Reading variables from memory does not change them
- Values placed into a variable replace (overwrite) previous values:

Shortcut Assignment Operators

- We can use additional operators to calculate values and assign them to the variable on the left all in one statement
  - Known as *shortcut assignment operators*
- The general syntax is:

```
variable_opr_expression;
```

- Where *opr* is one of the five arithmetic operators: `+`, `-`, `*`, `/`, `%`
- For example, the following two statements create the same result:

```
x = x + 3;
x += 3;
```

- Shown below are the assignment operators with examples of how they are used:

Summary of Assignment Operators

Operator	Description	Example	Equivalent To
=	Assigns the value of the expression on the right to the variable on the left	`x = 3`	
+=	Adds the expression on the right to the variable on the left	`x += 3`	`x = x + 3`
-=	Subtracts the expression on the right from the variable on the left	`x -= 3`	`x = x - 3`
*=	Multiplies the expression on the right to the variable on the left and saves the result in the variable on the left	`x *= 3`	`x = x * 3`
/=	Divides the variable on the left by the expression on the right and saves the result in the variable on the left	`x /= 3`	`x = x / 3`
%=	Calculates the remainder from dividing variable on the left by the expression on the right and saves the result in the variable on the left.	`x %= 3`	`x = x % 3`

Increment and Decrement Operators

- Adding or subtracting one is a common operation in programming
- C++ provides arithmetic shortcuts for these operations with the increment and decrement operators
- The *increment operator* (`++`) adds 1 to a variable's value
- *Pre-increment* adds 1 before evaluating an expression

```
++sum;
```

- *Post--increment* evaluates the expression and then adds 1

```
sum++;
```

- The decrement operator works like the increments operator, except it subtracts 1 from the variable:

```
--sum
sum--
```

- Pre- and post-- increment matters when the operation is part of a larger expression
- For example, consider the code:

```
int x = 5;
int y = x++;
cout << "x=" << x << " y=" << y;
```

- We may expect y to be 6 after this code executes
- Instead, y has the value of 5
- The reason is that ++ after a variable *(post-increment* is equivalent to:

```
y = x;
x = x + 1;
```

- On the other hand, ++ before a variable (*pre-increment*) is equivalent to:

```
x = x + 1;
y = x;
```

Casting

Cast - change the data type of the returned value of an expression

- Recall that different data types are stored in different forms
- Sometimes you need to change from one form to another
- For example: arithmetic adding a `double` and an `int` value
- C++ will *automatically* cast one value to another
  - Known as *implicit casting* or *type coercion*
- Programmers can also explicitly cast data types
  - *Explicit casting* changes the data type for a single use of the variable
- Precede the variable name with the new data type in parentheses:

```
(datatype) variablename
```

- The type is changed only for the single use of the value
- For example:

```
double a = 2.99999;
a += (int) a;
cout << a << endl;
```

- The value of `a` is converted from type `double` to `int` before assigning the converted value to `a`
- However, `a` remains a type `double` and the cast only applies to a single use of `a`
- The above example shows a common use of casting -- removing the decimal part of a floating-point number
- Note that the decimal portion of the number is truncated and NOT rounded
- Decimal part is lost (discarded, ignored, thrown away)
- Another use is to convert an `int` to a `double` when dividing two int numbers and a decimal result is desired
- For example:

```
double x = (double) 9 / 5;
cout << x << endl;
```

- Still another use is to prevent compiler warnings
- For example:

```
double a = 2.3;
int i = a;
cout << i << endl;
```

- The above may cause a compiler warning with the settings we use:

```
warning: converting to "int" from "double"
```

- To remove the warning, we use a cast:

```
double a = 2.3;
int b = (int) a;
cout << b << endl;
```

- This tells the compiler that you intended to convert from `double` to `int`

Example Application Using Casting:

```
1   #include <iostream>
2   using namespace std;
3
4   int main() {
5       double input;
6       cout << "Enter hours: ";
7       cin >> input;
8
9       int hours = (int) input; // prevents warning
10      int minutes = (int) ((input - (int) input) * 60);
11      cout << "It hours and minutes, this is "
12           << hours << ":" << minutes << endl;
13      return 0;
14  }
```

Integer Overflow

- An integer is stored in a computer as a pure binary number:

Image source: Dan Cassidy

- The sign bit sets whether the number is positive (0) or negative (1)
- The other bits represent the value, in this case `123`.
- There are only a finite set of numbers in an integer value
- What happens when an integer is too big for its type?

```
int bigPlus = 2147483647;
cout << "big number: ";
cout << bigPlus + 1 << endl;
int bigMinus = -2147483647;
cout << "Small number: ";
cout << bigMinus - 2 << endl;
```

- The number "goes around" from the highest number to the lowest
- You must be careful that your program will not go beyond the range of its data types
- Can't sleep - from xkcd

More Integer Types

- To increase the range, C++ has the `long` data type
- Originally, the `long` data type was 32 bits while the `int` was 16 bits
- However, with the development of 32 bit computers, the `int` value was extended to 32 bits but the `long` was left at 32 bits
- Thus, at the present time, `int` and `long` are the same size on most computers
- In addition, C++ has `unsigned` integer types you can use for the data range
- Rather than integer ranges from -2147483647 to 2147483647, `unsigned int` ranges from 0 to 4294967295
- New to C++11 is newer version of C++ is the type `long long` which is a 64 bit type.

Floating-Point Precision and Range

- Floating-point numbers are not exact representations of real numbers
- Rounding errors occur in repeated calculations
- Type double has about twice the precision of type float
- However, even type double can have rounding errors

```
cout << setprecision(17);
cout << .9F + .1F << endl;
cout << .8 + .2 << endl;
```

- When floating point numbers get too large, they are set to `inf`
- For instance:

```
cout << 2E38F + 2E38F << endl;
```

- Similarly, when numbers are too small they are set to `0.0`

The Moral

- Integer and floating-point data types work well most of the time
- However, if we work with large positive or negative integers, we must be sure we do not exceed the range of the data type
- Also, floating-point numbers have limited precision
- When math operations are performed repeatedly, they can become less precise
- Thus we must be careful of precision when using floating-point numbers

Activity 10.1: Prices (10pts)

- Find a partner for pair programming.
- Open up Eclipse and create a new projected named *Prices* with a file called *price.cpp*. Copy and paste the below code into it beneath the comment with your and your partner's names.

```
1   #include <iostream>
2   #include <iomanip>
3   using namespace std;
4
5   int main() {
6       string name;
7       double price = 0;
8
9       cout << "Enter the product name: ";
10      cin >> name;
11      cout << "Price of the " << name << ": ";
12      cin >> price;
13
14      // Insert new statements here
15
16      cout << "Total price: $" << price << endl;
17
18      return 0;
19  }
```

- Compile and run the starter program to make sure you entered it correctly.

When you run the program, the output should look like this:

```
Enter the product name: iPod_Nano
Price of the iPod_Nano: 149.50
Total price: $149.5
```

Note the format of the numbers output for the total price. We will address this formatting issue later in the exercise.

- Run the program again for a product with a very high cost, like a Boeing 777:

```
Enter the product name: Boeing_777
Price of the Boeing_777: 312345678
Total price: $2.12346e+08
```

Note the format of the numbers output for the total price. This format is called exponential notation. You may have encountered it in some of your math classes.

- Let us correct the formatting of the total price. Enter the following code before the statement that prints the total price:

```
cout << fixed                  // fixed notation, not scientific
     << setprecision(2);       // show 2 decimal places
```

These statements are referred to as the "magic formula" because they for C++ to output statements in a "standard" format. Note what each statement accomplishes.

- Compile and run your program again and verify the output looks like:

```
Enter the product name: Boeing_777
Price of the Boeing_777: 312345678
Total price: $312345678.00
```

- Let us add a constant that we will use later in our program. Enter the following code after the magic formula and before the statement that prints the total price:

```
const int PERCENT = 100;
```

A constant variable (or constant) is a variable that cannot change after being assigned a value. Using a constant lets us avoid using a *magic number*.

- Now we will add sales tax to the price of the product. Enter the following code after the constant and before the statement that prints the total price:

```
double taxRate = 0;
cout << "Enter sales tax rate (%): ";
cin >> taxRate;
double tax = price * taxRate / PERCENT;
price += tax;
```

Notice the last statement: `price += tax;`. This is an alternate way to code the statement: `price = price + tax;`.

- Compile and run your modified program and verify the output looks like:

```
Enter the product name: iPod_nano
Price of the iPod_nano: 89.50
Enter sales tax rate (%): 9.5
Total price: $98.00
(a whole dollars: $98
```

- Now we will find the whole dollars and cents of the amount to demonstrate *casting*. Enter the following code after the statement that prints the total price and before the return statement:

```
int dollars = (int) price;
cout << "In whole dollars: $" << dollars << endl;
```

Notice the `(int)` in the first statement. This is known as a *type-cast* or just *cast*.

- Compile and run your modified program and verify the output looks like:

```
Enter the product name: iPod_nano
Price of the iPod_nano: 89.50
Enter sales tax rate (%): 9.5
Total price: $98.00
In whole dollars: $98
```

Upcoming Assignments

- Assignment 9 due Tuesday at 9:20am
- Lab 5 due Friday

~Have a Great Evening!~