

PY410 / 505

Computational Physics 1

Salvatore Rappoccio

C++: Expressions and Arithmetic

- An expression is a sequence of operators and operands that specifies a computation
- Arithmetic is just like in regular math, but can happen on other types besides numbers!

C++: Expressions and Arithmetic

- We've already seen the standard OUTPUT in C++ (`cout`)
- Now to take a look at standard INPUT in C++ (`cin`)
- We will use `cin` to get two values and compute their sum
 - Enter them in order with a space between
- Go to piazza.com and copy the code from “Addition” into a file named “adder.cpp”

```
#include <iostream>
int main() {
    int value1, value2, sum;
    std::cout << "Please enter two integer values: ";
    std::cin >> value1 >> value2;
    sum = value1 + value2;
    std::cout << value1 << " + " << value2 << " = " << sum
    << '\n';
}
```

- Then compile and execute

C++: Expressions and Arithmetic

- What are we looking at? Individual expressions **ALWAYS** evaluate to a value
- Examples:

```
42;                      // value: 42
sum = value1 + value2;    // value: "sum"
12 > 13;                  // value: false
```

C++: Expressions and Arithmetic

- Arithmetic operators behave basically how you expect

Operator	Meaning
+	addition
-	subtraction
*	multiplication
/	division
%	modulus

- Can have **BINARY** operators (two operands) or **UNARY** operators (one operand)
 - For arithmetic operators, only “+” and “-“ can be unary

Logical versus Bitwise Operators

- Some confusion about “logical” versus “bitwise” operators
- Logical operators work on individual boolean variables:

e_1	e_2	$e_1 \&& e_2$	$e_1 e_2$	$!e_1$
false	false	false	false	true
false	true	false	true	true
true	false	false	true	false
true	true	true	true	false

- Bitwise operators do the same thing, bit by bit:
 - and ($\&$)
 - or ($|$)
 - exclusive or ($^$)
 - bit shift left ($<<$)
 - bit shift right ($>>$)

Logical versus Bitwise Operators

- bitwise comparison of “5” and “4”:

```
#include <iostream>

int main(void) {
    unsigned int i = 0x5;
    unsigned int j = 0x4;
    unsigned int k = i | j;
    unsigned int l = i & j;
    unsigned int m = i ^ j;
    std::cout << "i    = " << std::hex << i << std::endl;
    std::cout << "j    = " << std::hex << j << std::endl;
    std::cout << "i|j = " << std::hex << k << std::endl;
    std::cout << "i&j = " << std::hex << l << std::endl;
    std::cout << "i^j = " << std::hex << m << std::endl;
    return 0;
}
```

- Output:

```
i    = 5
j    = 4
i|j = 5
i&j = 4
i^j = 1
```

C++: Expressions and Arithmetic

- Example: go to piazza.com and get the code from “Arithmetic Operators”

```
#include <iostream>
int main(void) {

    std::cout << -5 << std::endl;
    std::cout << 5 + 3 << std::endl;
    std::cout << 5 * 3 << std::endl;
    std::cout << 21.32 / 38.0 << std::endl;
    std::cout << 12 / 4 << std::endl;
    std::cout << 13 / 4 << std::endl;
    std::cout << 13. / 4. << std::endl;
    return 0;
}
```

- Compile and execute, and answer:
 - Which of these does not behave the same as you would expect?
 - What is the difference between the last two expressions?

C++: Expressions and Arithmetic

- Arithmetic has to be done on TYPES
 - Types remain constant throughout the operation!
- Examples:
 - “int + int = int”
 - “int - int = int”
 - “int * int = int”
 - “int / int = int”
- But wait! The last one is dodgy... fractions are NOT integers!
 - Integers are NOT CLOSED under division!

C++: Expressions and Arithmetic

- So how do we handle integer division?
 - The same way ALL division is handled in C++

Truncation, truncation, truncation

- As integers:

$$9 / 3 = 3$$

$$10 / 3 = 3$$

$$11 / 3 = 3$$

$$12 / 3 = 4$$

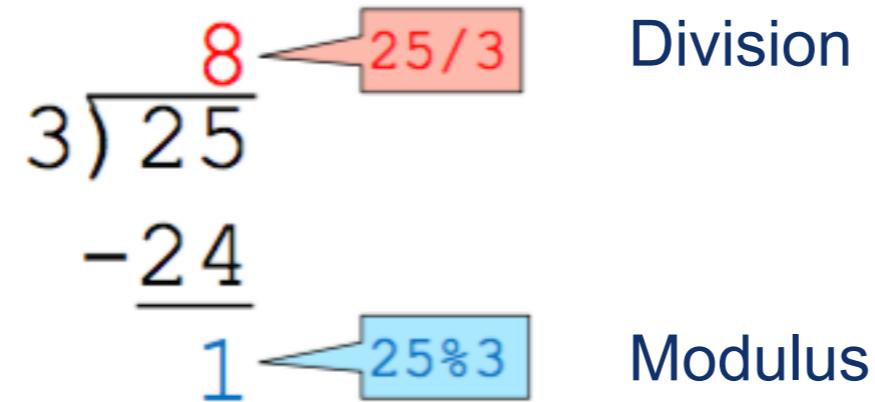
C++: Expressions and Arithmetic

- Integer division and modulus:

$$\begin{array}{r} 8 \\ 3 \overline{) 25} \\ -24 \\ \hline 1 \end{array}$$

Division

Modulus



- Division gives you the number of times the divisor (3) evenly goes into the dividend (25), i.e. 8
- Modulus gives you the remainder, i.e. 1
- Can be used in lots of applications (like, arrays! more later on that)

C++: Expressions and Arithmetic

- If you want ratios and fractions, you need floats or doubles!
- This is why “13 / 4” is different from “13. / 4.”
- $13/4$ gives you 3 (int)
- $13./4.$ gives you 3.25 (float or double)
- What about MIXED TYPE? $13. / 4 = ?$
- Go to piazza.com, download mixed.cpp from the “Mixed Types” entry.

C++: Expressions and Arithmetic

- Integers are a subset of reals
 - Therefore “int” can always be converted to “float” or “double”
 - The way we say this is int is “narrower” than float, and float is “wider” than int
- However, the converse is NOT true: this is called “narrowing”
 - Cannot represent 1.9 as an int
- The C++ standard says : TRUNCATION, TRUNCATION, TRUNCATION
 - It does NOT round!!!
 - “int i = 1.999999” gives you “1”, not “2”
 - Some compilers will warn you (“potential loss of data”)
 - Other compilers will happily give you the garbage you asked for.

C++: Expressions and Arithmetic

- If you have an expression with MIXED TYPES, the standard will “widen” the narrower one
 - so “float / int” will give you a “float”
 - Also “int / float” will give you a “float”
 - But remember “int / int” will give you an “int”

C++: Expressions and Arithmetic

- This is a whole lot of guessing, though
- Better way is called “casting”
 - What we did before was **IMPLICIT** casting
 - We now **EXPLICITLY** cast
- Several casting cases are possible, but we will focus on the first one now: “**static_cast**”.

```
int j = static_cast<int>( g );  
std::cout << j << std::endl;
```

- This says “interpret g as an integer, assign it to j”.
- We will go through other casts later
- **static_cast** is better because it can be checked at **COMPILE TIME** (very beneficial later on)

C++: Expressions and Arithmetic

- Operator precedence and associativity:
 - Follows same rules you've always learned:

“Please Excuse My Dear Aunt Sally”

=

Parentheses, Exponentiation, Multiplication, Division,
Addition, Subtraction

- Associativity also follows this
- But! Use parentheses to be clear when necessary!
 $f = 2 + 3*4;$
and
 $f = 2 + (3*4);$
- Both correct, but second is clearer

C++: Expressions and Arithmetic

- Formatting and whitespace : C++ does not care about either. All of these are okay:

```
#include <iostream>

int f1(void){ return 1;}
int f2(void){
    return 2;
}

int
f3
(void)
{
return
    3;
}

int main(void){

    std::cout << f1() << std::endl;
    std::cout << f2() << std::endl;
    std::cout << f3() << std::endl;

    return 0;
}
```

C++: Expressions and Arithmetic

- CANNOT put whitespace in between variable names or within an operator
- MUST have whitespace between type and variable name.
- These are OK:

```
int my_int = 0;  
float MyFloat = 0.0;
```
- These are not:

```
double My Double = 0.0;  
charMyChar('a');
```

C++: Expressions and Arithmetic

- Types of errors you can see:
 - Compile time : does not compile correctly
 - Run time: does not ALWAYS run correctly (ugh)
 - Logic error: compiles, runs, and gives you exactly what you told it, but you told it something wrong

C++: Expressions and Arithmetic

- “Shortcut” operators and “optimization” operators
- There are other operators that are shorthand for a combination
- Example: Incrementing a value:

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

- Can also be written as
- ```
x++;
```
- OR!
- ```
++x;
```
- “Post-increment” and “pre-increment” operators
- Also have “minus minus”

C++: Expressions and Arithmetic

- Post-increment versus pre-increment:
 - Post: increment AFTER statement is executed
 - Pre: increment BEFORE statement is executed
- If just alone, no difference. These are equivalent:

```
int x = 0;  
x = x + 1;  
x++;  
++x;
```

- If inside more complicated statement, there is a difference:

```
int x1 = 1;  
int x2 = 1;  
int y1 = ++x1;  
int y2 = x2++;
```

```
std::cout << "x1 = " << x1 << ", y1 = " << ++x1 << std::endl;  
std::cout << "x2 = " << x2 << ", y2 = " << x2++ << std::endl;
```

- gives:

```
x1 = 2, y1 = 2  
x2 = 2, y2 = 1
```

C++: Expressions and Arithmetic

- Bitwise operators also work (just like arithmetic):
 - and (&)
 - or (|)
 - exclusive or (^)
 - bit shift left (<<)
 - bit shift right (>>)

C++: Conditional Execution

- All CONDITIONS in C++ evaluate to bools
- Possible conditions:

Operator	Meaning
==	Equal to
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
!=	Not equal to

Table 5.1: C++ Relational operators

- Now you can see the boolean VARIABLES assigned to the output of EXPRESSIONS:

```
bool expression = 14 < 16;  
std::cout << expression << std::endl;
```

C++: Conditional Execution

- The execution can then be CONDITIONAL upon the outcome of a boolean variable
- Simplest format is the “if/else” formalism

C++: Conditional Execution

- This works exactly as you expect, but pedantically:

```
if ( condition )
```

```
    statement
```

- Example:

```
int i1 = 0;
if (i1 < 2) {
    std::cout << "i1 is too small. Eat more." << std::endl;
}
```

- Note:

- The statement is ONLY ONE STATEMENT

- If you want multiple lines, enclose in curly braces

C++: Conditional Execution

- For if/else:

```
if ( condition )
```

```
    statement 1
```

```
else
```

```
    statement 2
```

- Example:

```
int i1 = 0;
if (i1 < 2) {
    std::cout << "i1 is too small. Eat more." << std::endl;
}
else {
    std::cout << "i1 is big enough." << std::endl;
}
```

- Again: only SINGLE STATEMENTS come after, multiples must be in curly braces

C++: Conditional Execution

- Can nest or do sequences. Go to piazza.com, download code in “Sequences of Conditionals”:

```
#include <iostream>

int main(void) {

    int i = 0;
    std::cout << "Enter a number: ";
    std::cin >> i;

    if ( i > 2 ) {
        std::cout << "This is greater than 2. Way too much!" << std::endl;
    } else{
        if( i == 2 ) {
            std::cout << "Phew! This is 2." << std::endl;
        } else if (i == 1) {
            std::cout << "So close, but this is only 1!" << std::endl;
        } else {
            std::cout << "Yuck, this is even less than 1." << std::endl;
        }
    }

    return 0;
}
```

C++: Conditional Execution

- But bools are just one single bit
- What if your expression gives you another type?
- Example:

```
#include <iostream>

int main(void) {

    int i = 0;
    std::cout << "Enter a number: ";
    std::cin >> i;                                Uhhh... what?

    if ( i + 5 ) { ←
        std::cout << "Tweet!" << std::endl;
    } else {
        std::cout << "Nuke!" << std::endl;
    }

    return 0;
}
```

C++: Conditional Execution

- If your expression is cast to “0” (zero), then it is false
- If your expression is cast to “!0” (not zero), then it is true
- So this looks like one thing, and gives you something else you didn’t expect, but it is exactly what you told it to do:

```
#include <iostream>

int main(void) {

    int i = 0;
    std::cout << "Enter a number: ";
    std::cin >> i;

    if ( i = 5 ) { ←
        std::cout << "Nuke!" << std::endl;
    } else {
        std::cout << "Tweet!" << std::endl;
    }

    return 0;
}
```

As long as this is
not equal to 5,
we're saved???

C++: Conditional Execution

- Logic operations are very useful here also:
- Like bitwise, but TWO symbols together
 - Logical and: &&
 - Logical or : ||
 - Logical not: !
 - Logical xor: ^^

C++: Narrowing

- Go to “[piazza.com](#)” and download “narrowing.cpp”:

```
int main(void) {  
    double d = 22000000000000.0;  
    int i = d;  
    std::cout << "d = " << d << ", i = " << i << std::endl;  
    return 0;  
}
```

- Now compile with the “-Wconversion” flag (enables conversion... don’t ask me why -Wall didn’t work last time):

```
> g++ -Wconversion narrow.cpp -o narrow
```

- and you get:

```
narrow.cpp: In function ‘int main()’:  
narrow.cpp:9: warning: conversion to ‘int’ from ‘double’ may  
alter its value
```

- And sure enough, if you try to run:

```
d = 2.2e+13, i = -2147483648
```

C++: Floating point comparison

- We've seen the “`==`” operator for ints
 - If we try “`5 == 5`”, it returns “true”
 - If we try “`1 == 0`”, it returns “false”
 - (unless you're KellyAnne Conway, in which case it returns “`alternative_true`”)
 - If we try “`5.0 == 5.0`”, what does this do?
 - What does this even mean?

C++: Floating point comparison

- Go to piazza.com again, get the “Floating point comparison” code

```
#include <iostream>

int main(void) {

    float f1 = 5.0f;
    float f2 = 5.0000000001f;

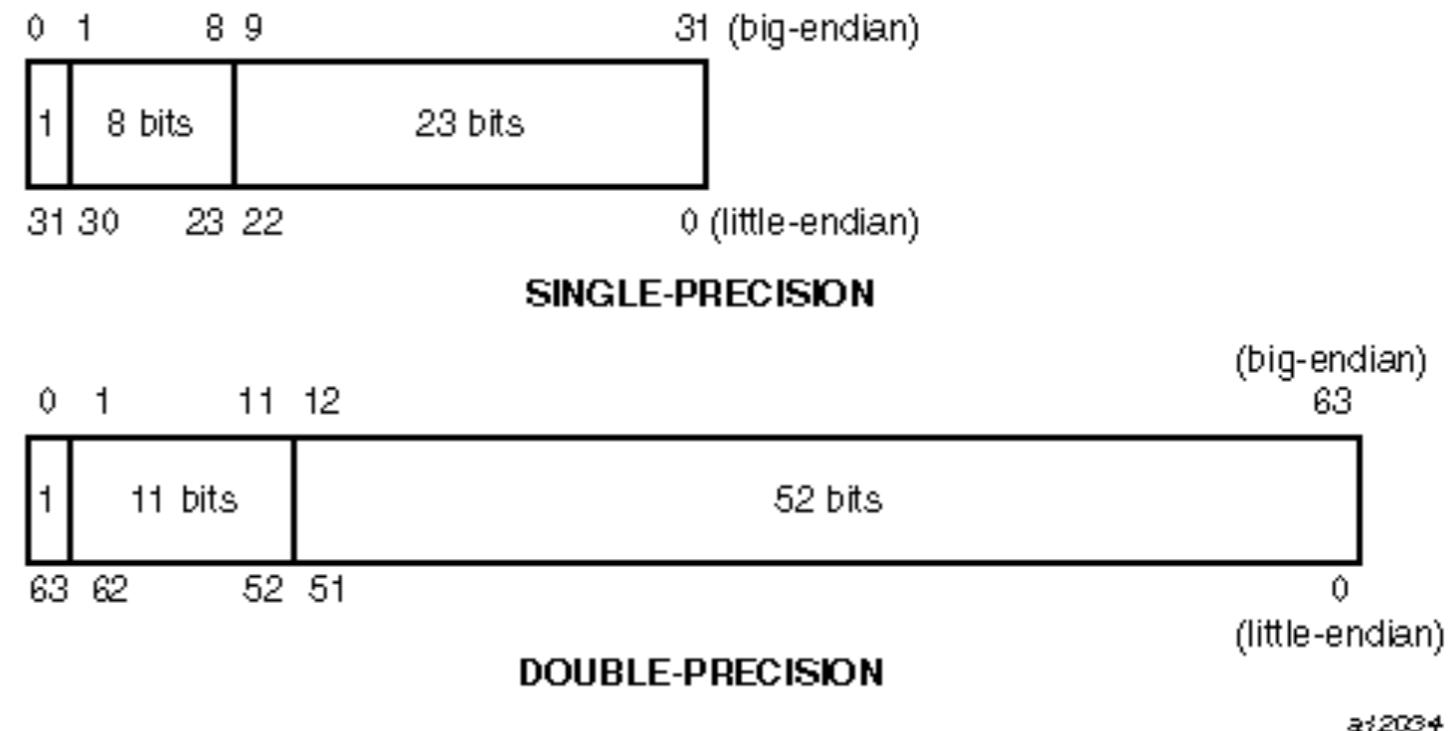
    if ( f1 == f2 ) {
        std::cout << "Nuke!" << std::endl;
    } else {
        std::cout << "Tweet!" << std::endl;
    }

    return 0;
}
```

- Compile and run, what do you get?

C++: Floating point comparison

- Comparing floats only makes sense within the precision of the “mantissa”!



- Even still, terrible idea to try the “`==`” operator

- Better: assign a tolerance you can live with, and look if it is within the tolerance!
 - BAD: “`f1 == f2`”
 - GOOD: “`std::abs(f1 - f2) < tolerance`”
- You need to pick a tolerance your program needs
 - For C++ “tolerance”, you can use
`std::numeric_limits<double>::epsilon()`

C++: Floating point comparison

- Go to piazza.com again, get the “Floating point comparison Improved” code

```
include <iostream>
#include <cmath>
#include <limits>

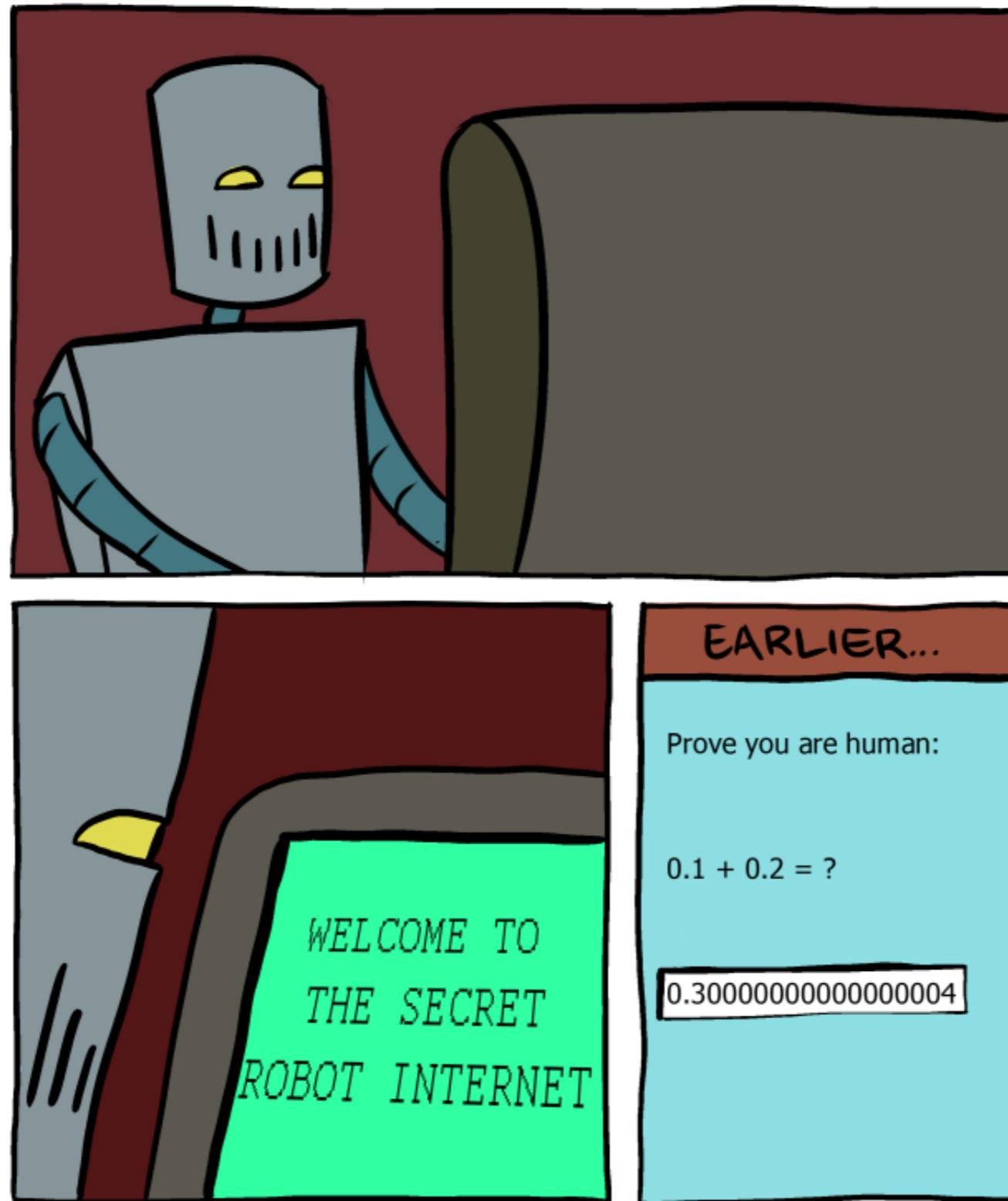
int main(void){
    float f1 = 5.0f;
    float f2 = 5.000000001f;
    float tolerance = 0.01f;

    if ( std::abs(f1 - f2) < tolerance) {
        std::cout << "Nuke!" << std::endl;
    } else {
        std::cout << "Tweet!" << std::endl;
    }

    if ( std::abs(f1 - f2) < std::numeric_limits<float>::epsilon() ) {
        std::cout << "Within machine precision!" << std::endl;
    }
}
```

- Compile and run, what do you get?

C++: Floating point comparison



C++: Strings

- You may have noticed that there is nothing for a BUNCH OF CHARACTERS together in C++
- This is called a “string” in other languages
- C++ has no intrinsic concept of a “string”, it’s just a bunch of “characters” lined up
- We will go over strings in detail later, but there is a library called the “Standard Template Library” that we’ve already seen (**#include <iostream>**)
- Now we will use the “strings” from the standard template library (**std**)
- Strings can basically use the standard logical expressions as you expect, but we will go into more later

C++: Strings

- Go to piazza.com, download the code under “Strings”:

```
#include <iostream>
#include <string>

int main(void) {

    std::string s1;
    std::cout << "Enter a string: ";
    std::cin >> s1;

    std::cout << "Your string is: " << s1 << std::endl;

    if (s1 == "Yay!") {
        std::cout << "Yay? Just what I was thinking!" << std::endl;
    }
}
```

- Grad students: you can utilize something like this for your HW's

C++: Switch

- There is another option for multiple-way “if” statements: “Switch”
- Just like a giant “if/else” statement, but easier to use
- Constraint: can only use on integer types

```
switch ( integral expression ) {  
    case integral constant 1 :  
        statement sequence 1  
        break;  
    case integral constant 2 :  
        statement sequence 2  
        break;  
    case integral constant 3 :  
        statement sequence 3  
        break;  
    . . .  
    case integral constant n :  
        statement sequence n  
        break;  
    default:  
        default statement sequence  
}
```