Variables; Arithmetic; Output

ITP 165 – Fall 2015
Week 1, Lecture 2

Variables

- A *variable* allows us to store a value in a symbolic name that we can retrieve later

- Think basic algebra:

x = 10

y = x + 5

- Just like in algebra, you ***must*** declare a variable before you can use it in a subsequent expression

Type

- In C++, all variables ***must*** have a type

- A type tells us what sort of value is stored in the variable

- For example, it could be:
  - Text
  - A whole number
  - A decimal number
  - A true or false value

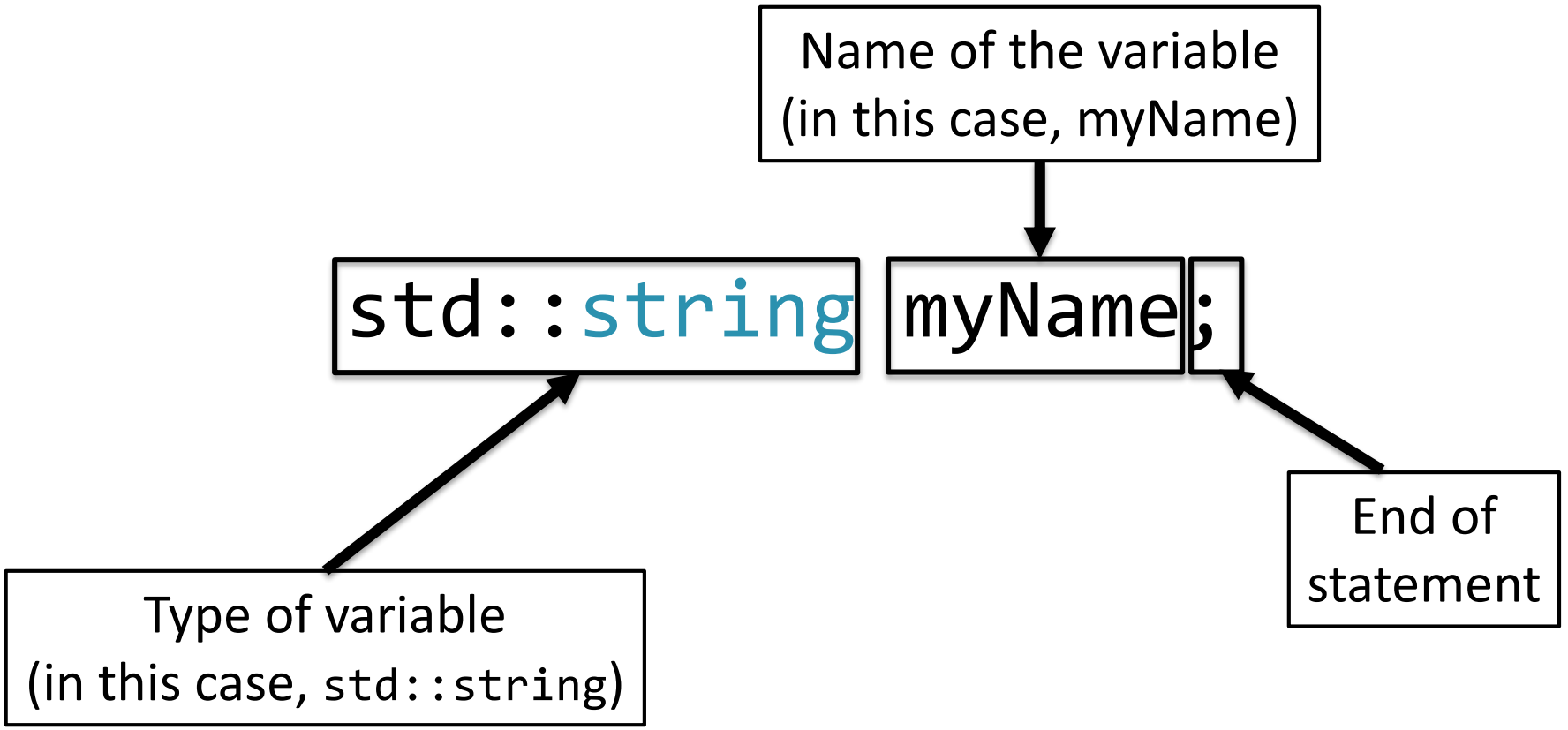
std::string

- The std::string type represents text

- Example program:

```
// Include string library
#include <string>
int main()
{
    // Declare a std::string
    std::string myName;
    return 0;
}
```

Variable Declaration Syntax



Special Text

- Some text cannot be input as they normally would
- Need special notation for certain text called ***escape-sequences***

Value	Description
\n	New line
\t	Horizontal tab
\v	Vertical tab
\b	Backspace
\r	Carriage return
\f	Form feed
\a	Alert
\\	Backslash
\"	Double quote

Escape Sequences

- If we wanted to add a horizontal tab

```
#include <string>
#include <iostream>
int main()
{
    std::cout << "This is left aligned text." << std::endl;
    std::cout << "\tThis text is indented." << std::endl;

    return 0;
}
```

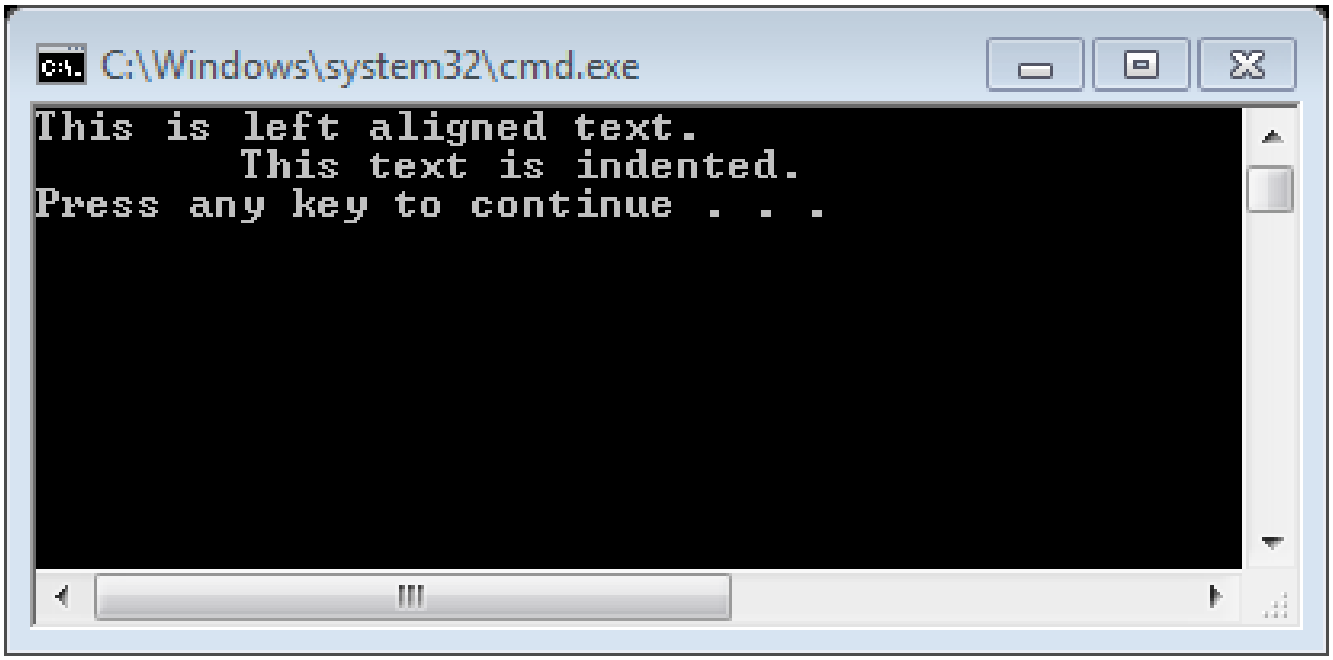
Escape Sequences

- If we wanted to add a horizontal tab

```
#include <string>
#include <iostream>
int main()
{
    std::cout << "This is left aligned text." << std::endl;
    std::cout << "\tThis text is indented." << std::endl;

    return 0;
}
```

Escape Sequences

- Result:



Variable Name Restrictions

- Variable names in C++:
  - Must begin with a letter or an underscore
  - Can be followed by any number of letters, numbers, or underscores
  - Cannot be named after language "keywords"

Valid	Invalid
`x`	`return`
`myName`	`123`
`__stuff`	`12myVar`
`Name2`	

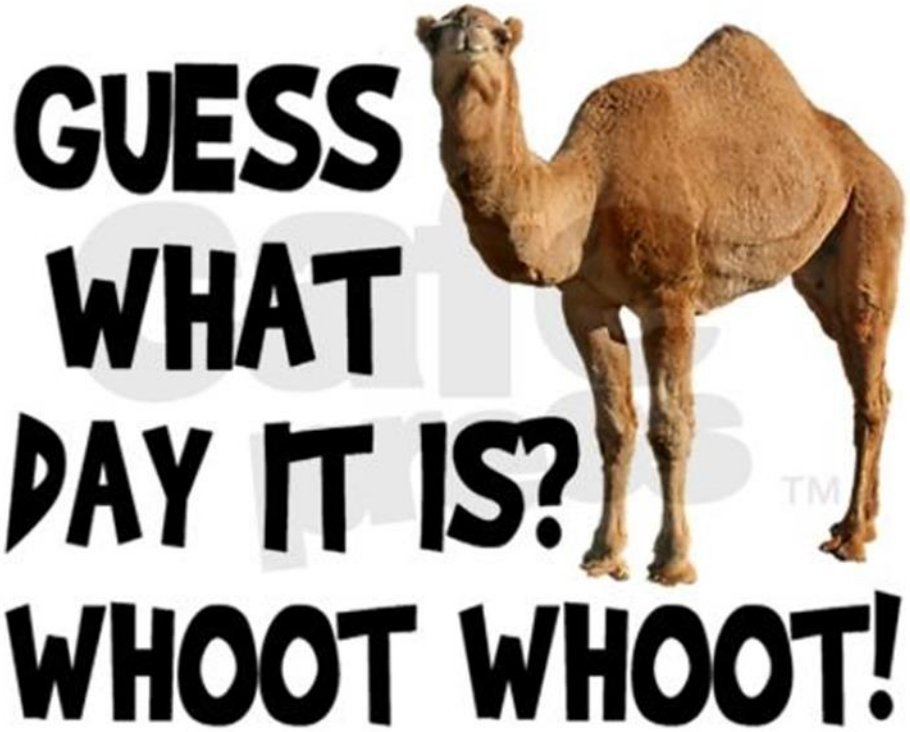
- That being said, you should ***always*** use meaningful variable names so it helps make your program easier to understand. `firstName` is a lot more descriptive than `x`.

Variable Name Restrictions

- In this class we will have a consistent naming convention for proper coding etiquette
- camelCase

- Rules of camelCase:
  - First letter ALWAYS lowercase
  - If name contains two or more words, the first letter of every word AFTER the first word is ALWAYS uppercase

- Examples:
  - firstName
  - variable
  - theBestExamplePossible



Variable Declaration with Assignment

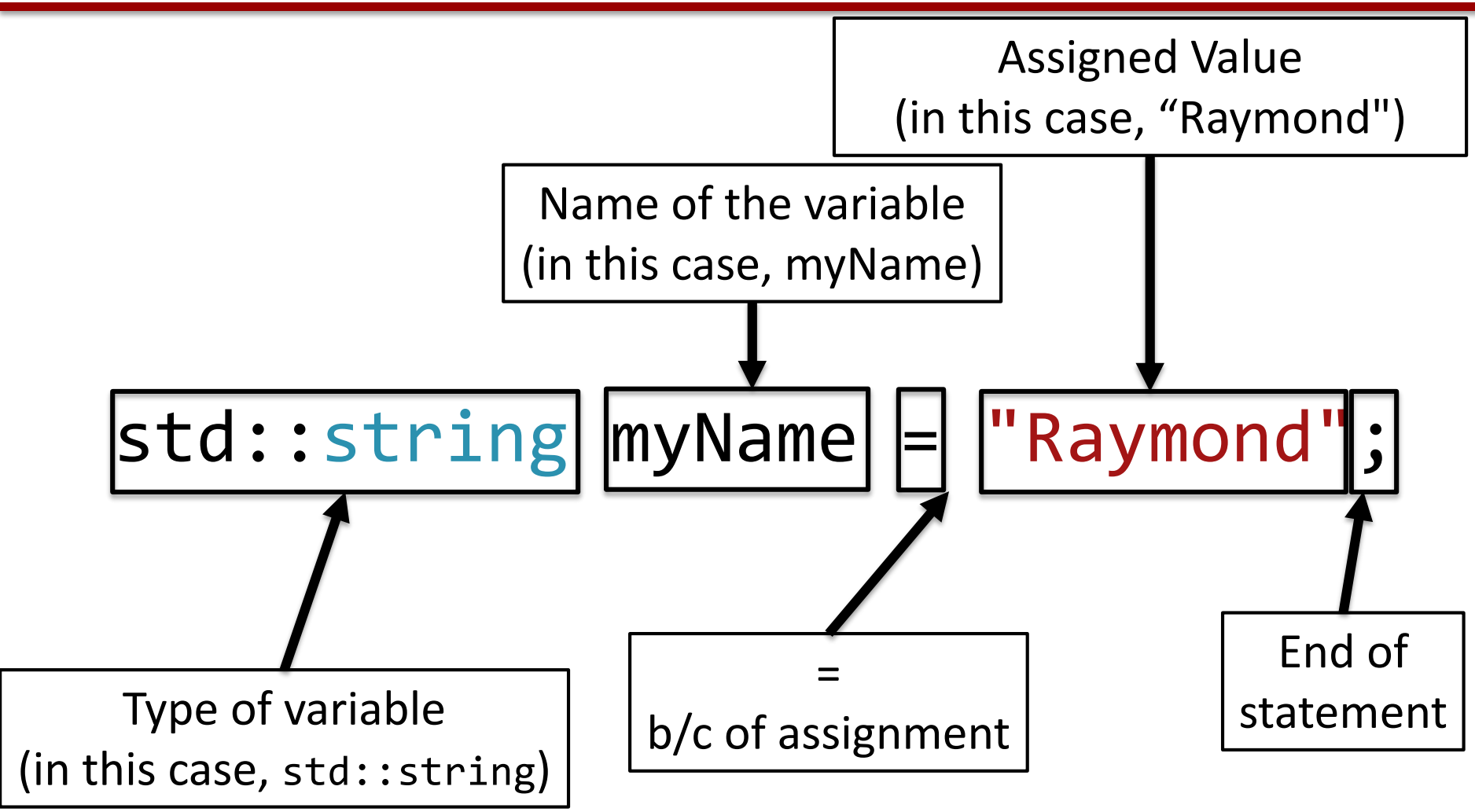
- It's a good practice to always initialize variables to a sane default value, for example, instead of:

```
std::string myName;
```

- It's preferable to say…

```
std::string myName = "Raymond";
```

Variable Declaration w/ Assignment Syntax



Reassignment

- You can later reassign a variable using similar syntax, except you don't specify the type on reassignment:

```
// Initialize to Raymond
std::string myName = "Raymond";
// Actually I want to change my name...
myName = "Fred";
```

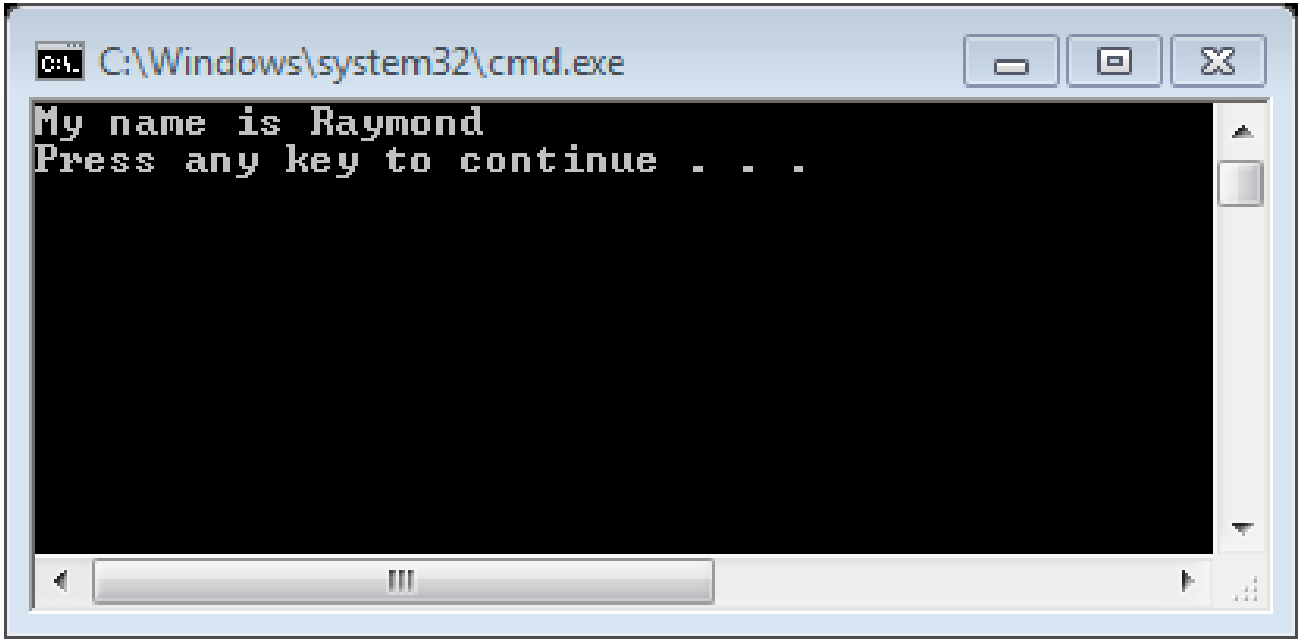
Outputting Variables

- We can output a variable within a cout, like:

```
// We have to include both iostream and string!
#include <iostream>
#include <string>
int main()
{
    std::string myName = "Raymond";
    // output the value of myName
    std::cout << "My name is " << myName << std::endl;
    return 0;
}
```

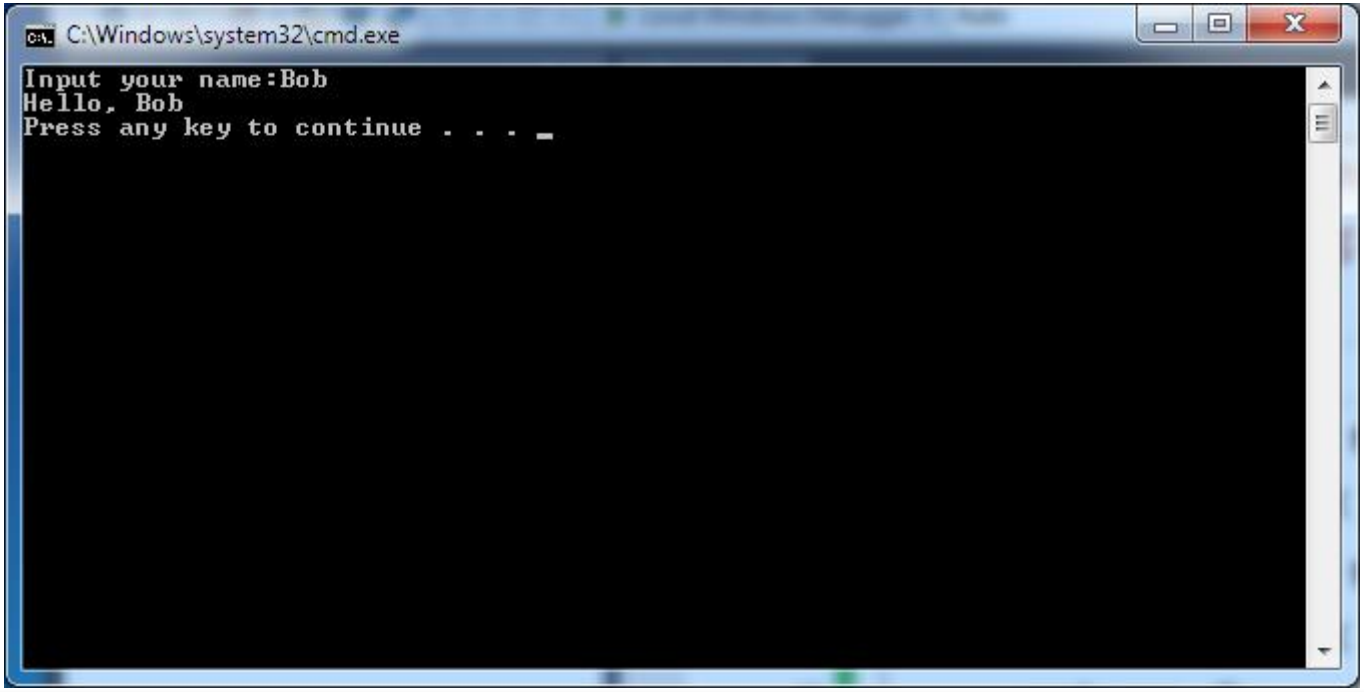
Outputting Variables, Cont'd

- When we run the program on the previous slide, we get:



std::cin

- std::cin is for console input
- We can have the user type in words into the program, and store them into our variables!
- For example:



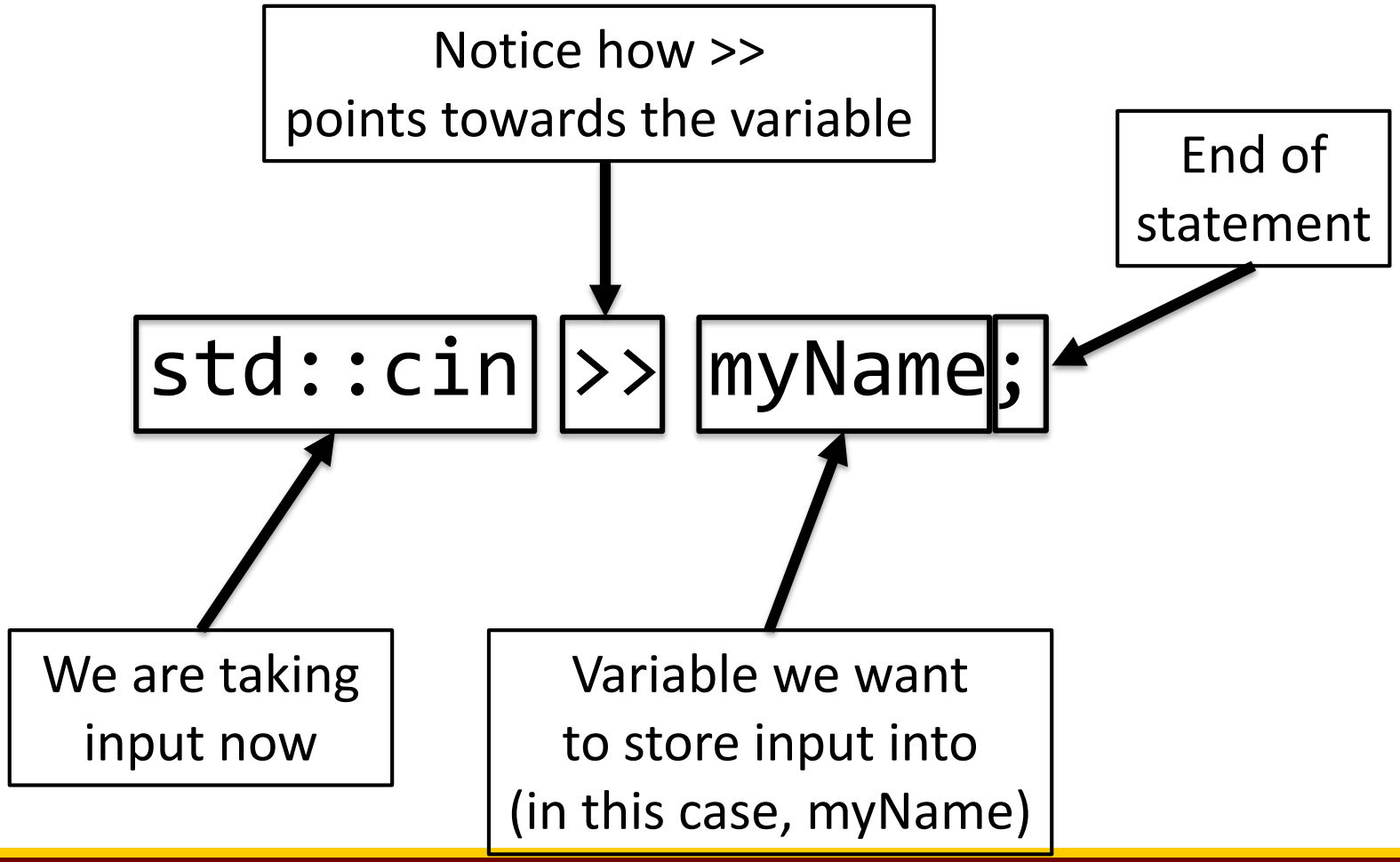
Code for previous program

```
#include <iostream>
#include <string>
int main()
{
    std::string myName;
    std::cout << "Input your name:";
    std::cin >> myName;
    std::cout << "Hello, " << myName << std::endl;
    return 0;
}
```

What's new?

```
#include <iostream>
#include <string>
int main()
{
    std::string myName;
    std::cout << "Input your name:";
    std::cin >> myName;
    std::cout << "Hello, " << myName << std::endl;
    return 0;
}
```

std::cin syntax



std::cin chaining

- You can technically chain std::cin (just like std::cout)

- However, I don't recommend doing this, it just makes this confusing

Numeric Types

- C++ supports both whole numbers and real (decimal) numbers
- Whole numbers – `int`
- Real numbers – `double`
- You need to specify which type of number a specific variable is, and the behavior will be different depending on the type you choose

Whole Numbers

- Most common whole number type is `int`
- (Abbreviation for integer)
- Can range approximately -2 billion to +2 billion*

- Example:

```
// Declare an integer x that's initialized to 12
int x = 12;

// Change x so it instead is 20
x = 20;
```

Commas and Numbers

- In C++, you do not use commas in numbers, ever

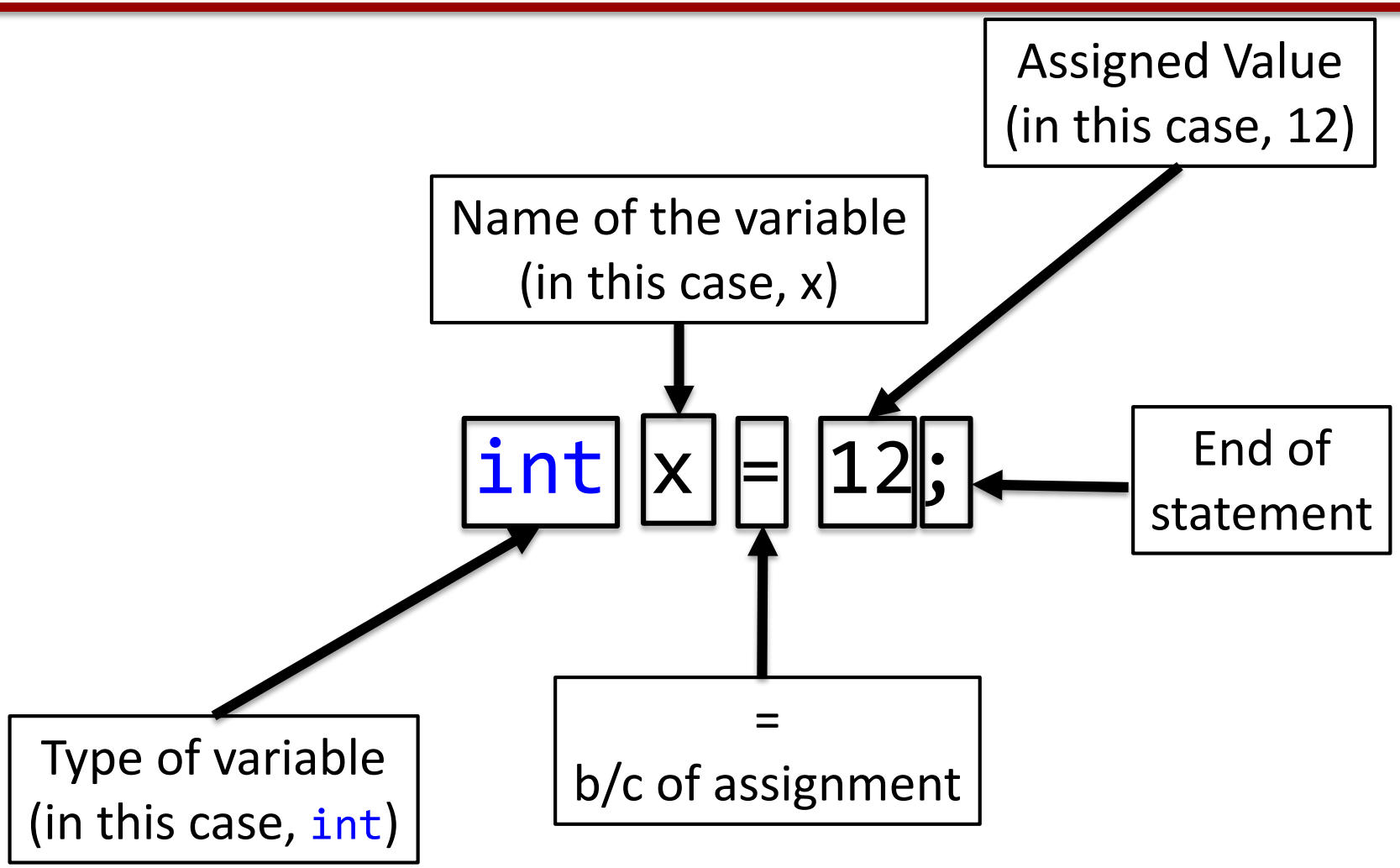
- So twenty-five thousand is:

```
25000
```

- ***NOT***

```
25,000
```

Variable Declaration w/ Assignment Syntax



Basic Arithmetic

- You can perform standard arithmetic using the following symbols or ***operators***

Operator	Math Performed
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulo (remainder of division)

Basic Arithmetic Examples

```
// Declare an int x that's set to 12
int x = 12;

// Declare an int num that's set to 7+3 = 10
int num = 7 + 3;

// Declare an int y that's set to num*x = 120
int y = num * x;
```

Division

- Division with integers may not work exactly as you might expect…

```
// What is x set to??
int x = 10 / 100;

// x = 0, because 10 / 100 is 0 remainder 10
```

- Division between integers is ***whole number division***, so there is no fractional part

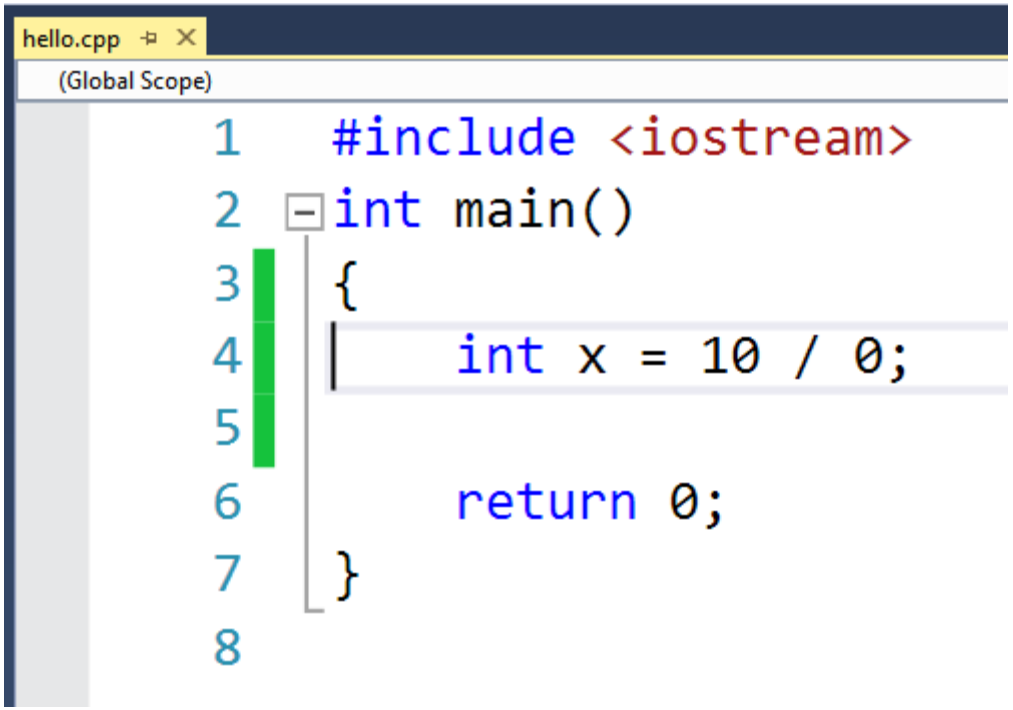
Division by Zero

- What happens if you try to write the following code?

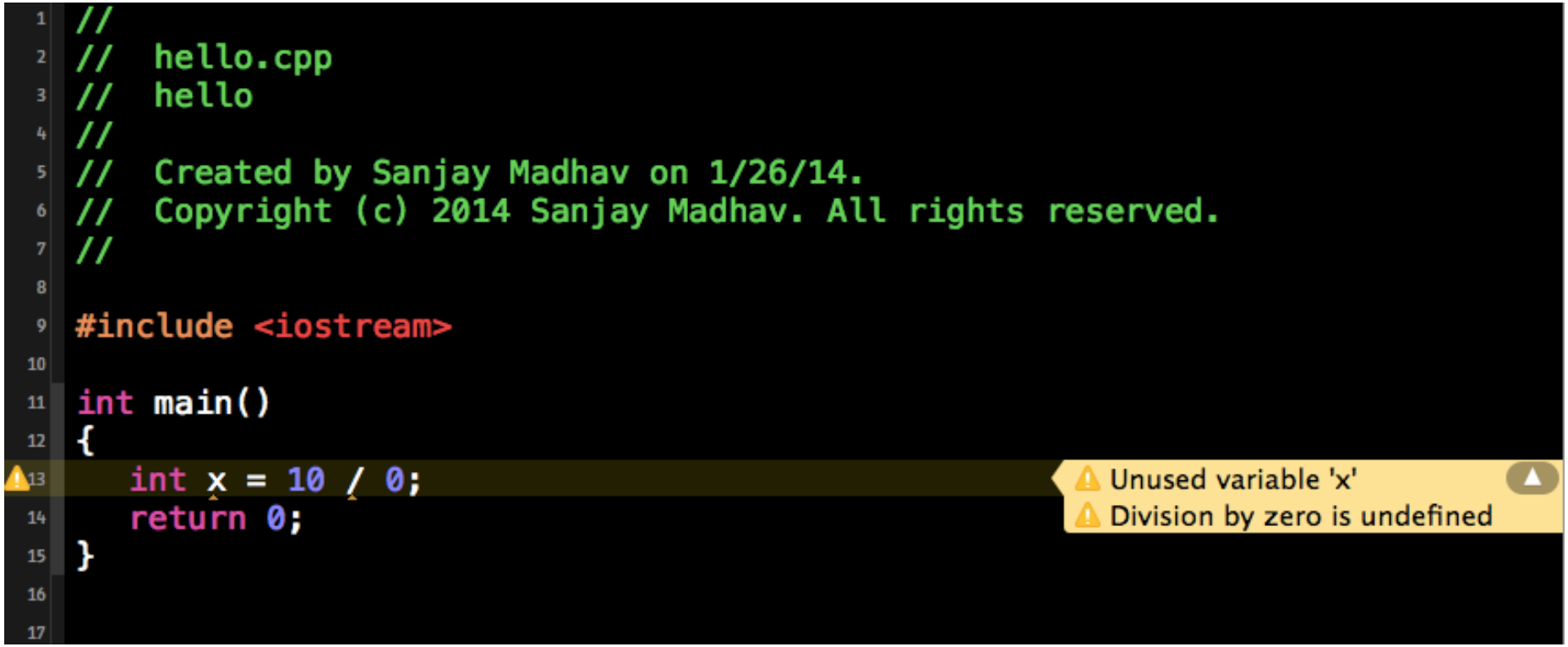
```
int x = 10 / 0;
```

Visual Studio Being Unhelpful

- Where are the red swigglies?



Xcode is More Helpful



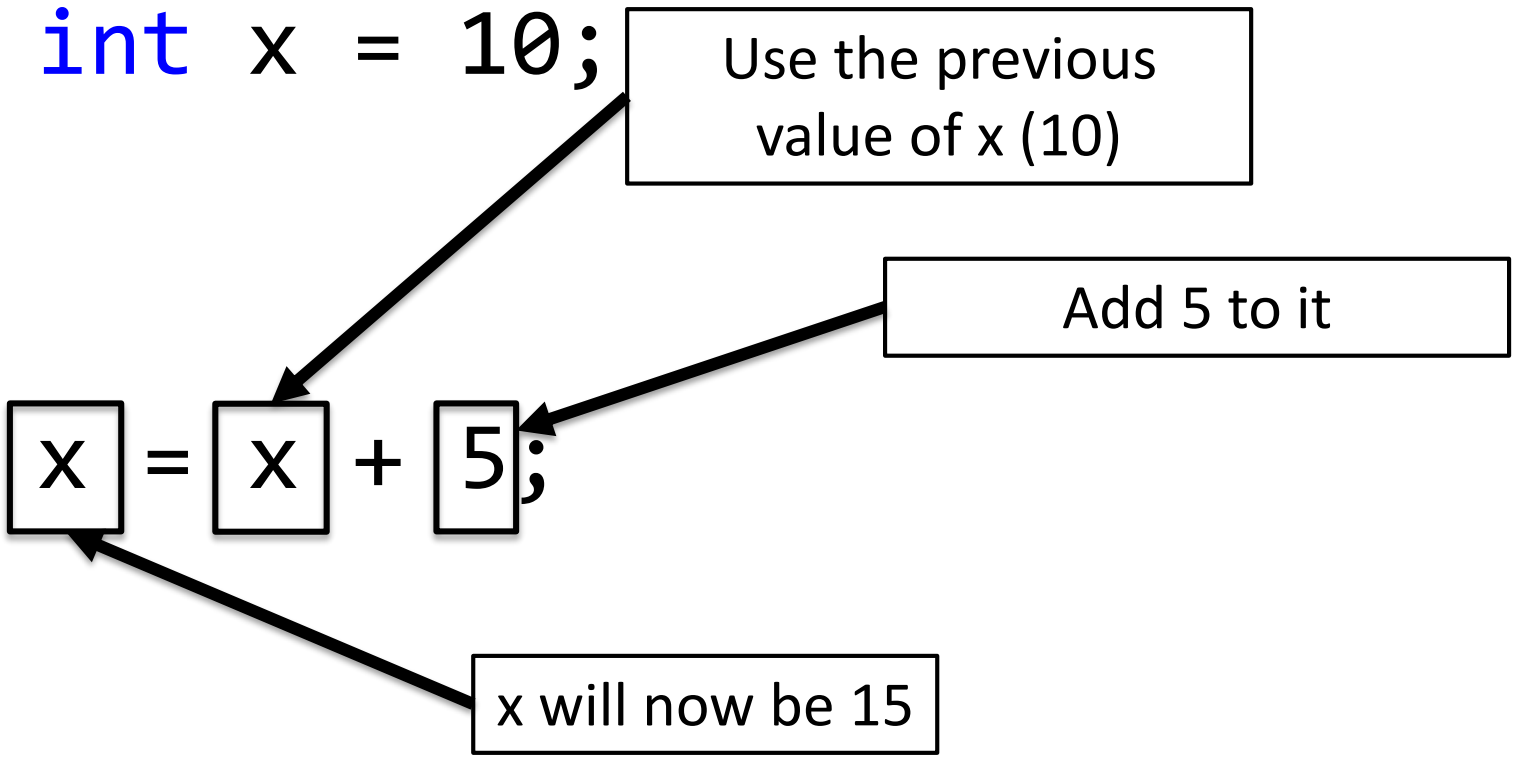
Modulo

- Modulo calculates the remainder, so for example:

```
// Sets x to 10
// (Because 10 / 100 is 0 remainder 10)
int x = 10 % 100;

// Sets y to 0
// (Because 25 / 5 is 5 remainder 0)
int y = 25 % 5;
```

Another Question

- What happens when you do this?



Reassignment Shortcut

- If you want to add back to the same variable, you can use shortcuts:

```
int x = 10;
// Below is the same as x = x + 5
x += 5;
```

- You can call the above operator “plus equals”*

Increment/Decrement

- If you want to just add one or subtract one from a variable and store it back, you can use a further shortcut:

```
int x = 10;

// Adds 1 to x, so x becomes 11
x++;

// Subtracts 1 from x, so x goes back to 10
x--;
```

Full int Operator Cheat Sheet

Operator	Math Performed
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulo
+=	Plus Equals
-=	Minus Equals
*=	Times Equals
/=	Divide Equals
%=	Modulo Equals
++	Increment
--	Decrement

Cin/Cout with int

- You can cin/cout with an `int` the same way as with a string:

```
#include <iostream>
int main()
{
    int x = 5;
    // Outputs 5, followed by an end of line
    std::cout << x << std::endl;
    // Outputs x * 2 which is 10, followed by an end of line
    std::cout << x * 2 << std::endl;

    int y = 0;
    // Stores the user input value into y
    std::cin >> y;
    return 0;
}
```

Parenthesis

- You can also use parenthesis to perform slightly more complex calculations:

```
// Sets x to 80
int x = (5 + 3) * 10;
```

- The term ***expression*** is used to generically refer to any values, whether they are specific numbers or result of calculations

Real (Decimal) Numbers

- C++ supports ***approximations*** of real (decimal) numbers

- This means that, for example, if you calculate:

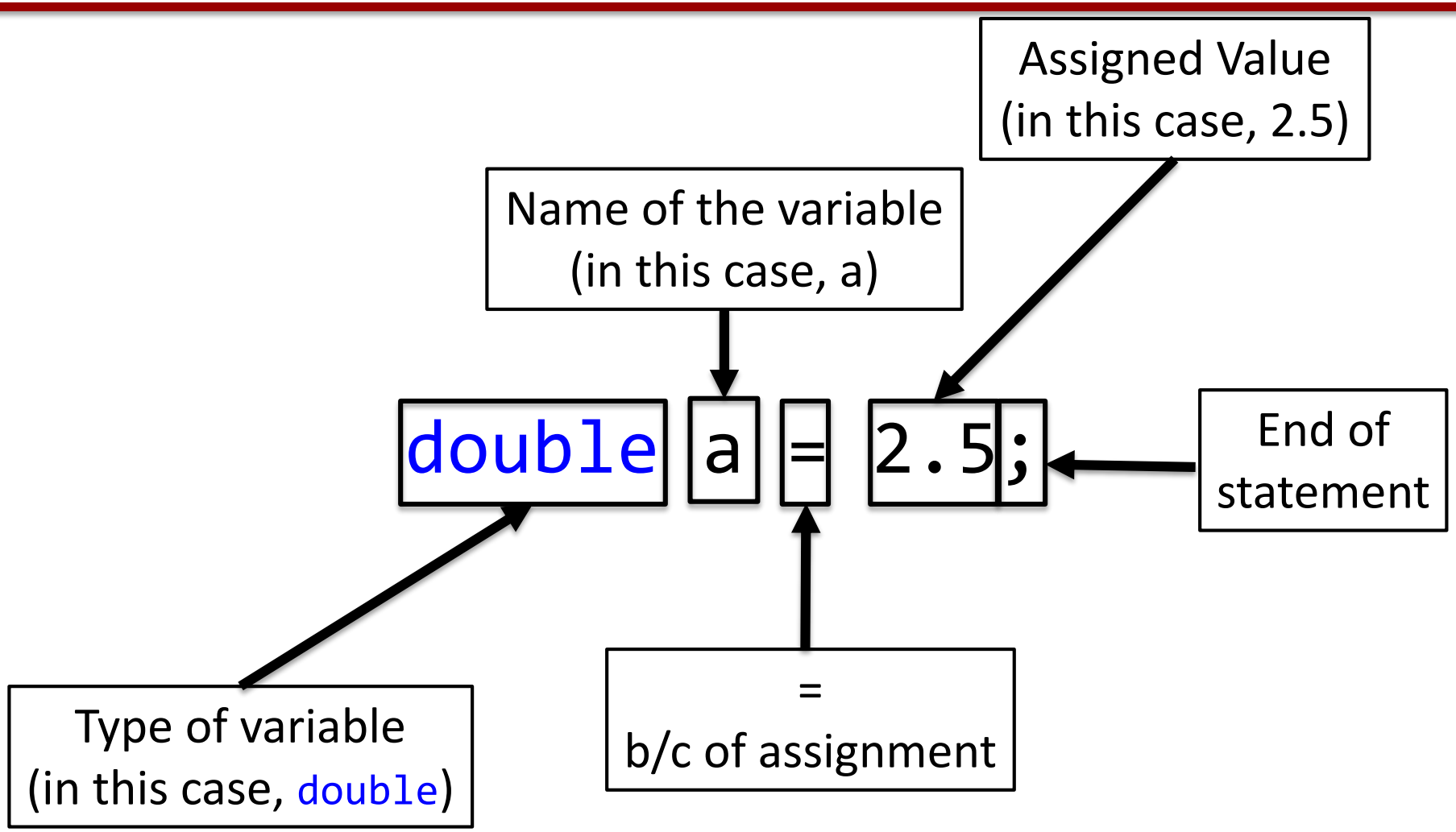
```
4.0 / 2.0
```

- …you are ***NOT*** guaranteed to get exactly 2.0. You may get 1.99999 or 2.00001

Double

- `double` is one real number type
- Can range from 1.7×10^{-308} to 1.7×10^{308}
- Has ~15 significant digits
- You will sometimes hear the term ***floating-point*** used to refer to computer approximations of real numbers

Variable Declaration w/ Assignment Syntax



Double Operator Cheat Sheet

The operators are the same, except ***there is no modulo***!

Operator	Math Performed
+	Addition
-	Subtraction
*	Multiplication
/	Division
+=	Plus Equals
-=	Minus Equals
*=	Times Equals
/=	Divide Equals
++	Increment
--	Decrement

Double Arithmetic Examples

- Between doubles, operators work exactly as you'd expect:

```
double a = 10.0;

// Declare/set b as ~4.0
double b = a / 2.5;

// Declare/set c as ~1.0
double c = a * 0.1;

// Add 1 to c, so ~2.0
c++;
```

Mixing and Matching int and double

- Sometimes, the results will be exactly as you expect…
- Example:

```
int x = 5;

double a = x;

double b = 13 / a;
```

x is an `int`, but it will be converted to a `double` (`~5.0`) and stored into a

13 is an `int`, but since we are dividing by a `double`, it will do real number division, and we'll end up with `~2.6` in `b`

Mixing and Matching int and double, cont'd

- Sometimes it will not give you what you might expect…

```
double a = 13 / 5;
```

13 and 5 are both `int`s, so it assumes you want whole number division, so 13/5=2.

Then, it converts 2 to a `double` (~`2.0`) and stores this in the variable a.

Fixing the Previous Issue

- The easy way is to just add a .0 to one of the numbers:

```
// This will store ~2.6 in a, because it will do
// real number division!
double a = 13.0 / 5;
```

But what if it's stored in an int variable?

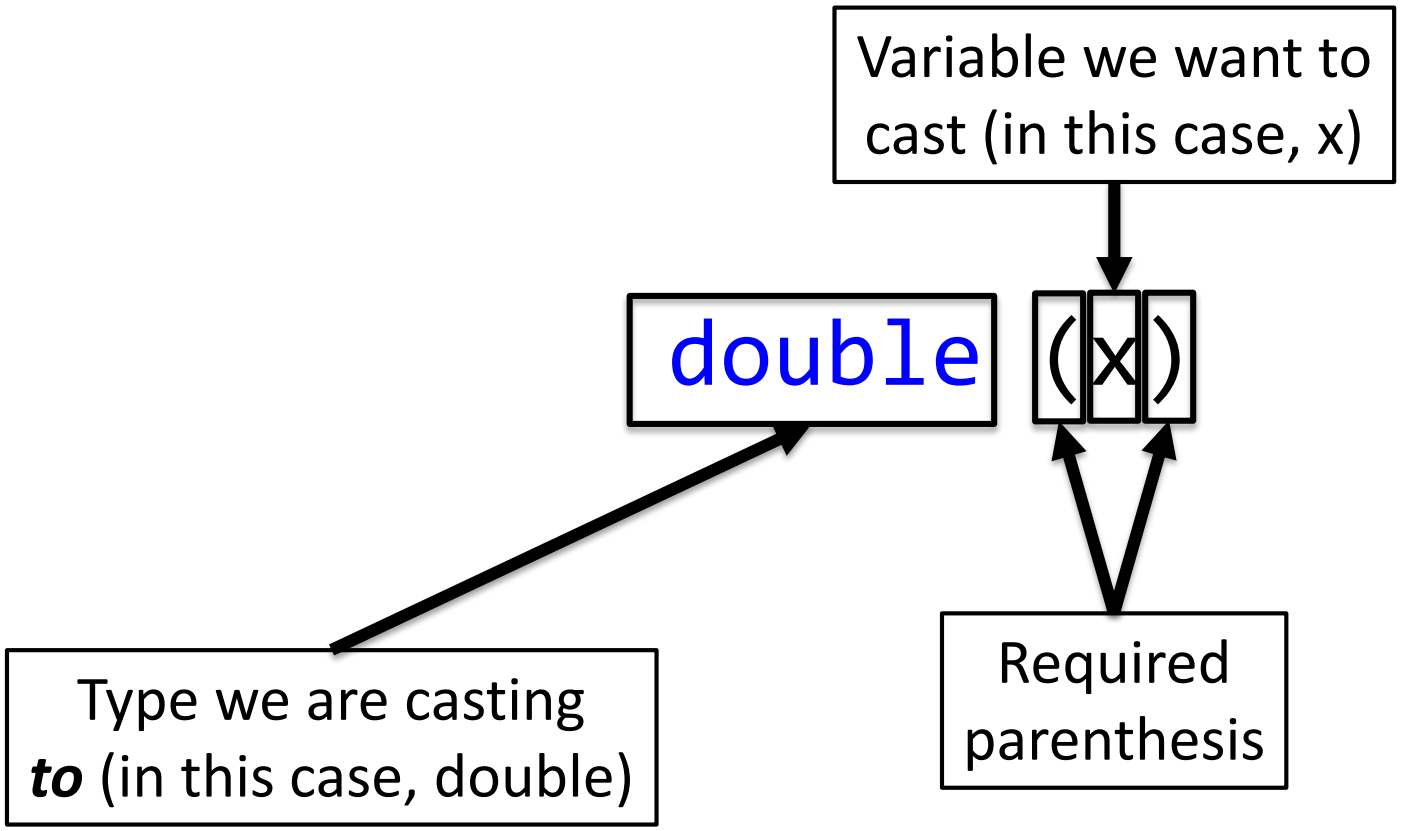
```
int x = 13;
int y = 5;

// This will store ~2.0 in a
double a = x / y;
```

Solution: Casting

- ***Casting*** allows us to forcibly convert from one type to another

```
// This says "treat x as a double"
// And so will store ~2.6 in a
double a = double(x) / y;
```

Casting Syntax



Other Numeric Types

- There are several other numeric types, as well, but don't worry about them for now

- Some examples…
- `short` – A smaller integer (+/- 32,000ish)

- `long long` – A bigger integer (+/- 9,223,372,036,854,775,808)

- `float` – A less accurate (but potentially faster) real number

Lab Practical #1: Mad Libs

- Let's write our own Mad Libs!