PIC 10A: Week 3a

Section 1C, Winter 2016
Prof. Michael Lindstrom (TA: Eric Kim)
v1.0

Announcements

- HW1 due this Wednesday (11 PM)
- Mini-midterm this Friday (lecture, 8 AM)
-

Today

- Variables
- More on Data Types
  - char, bool
  - static_cast
- User Input: cin
- Operator shorthands
  - x++, ++x, --x, x--
  - x+=1, x*=3, x/=5, x-=2
  - Modulo: x % y
- Numerical Issues
  - overflow/underflow, accuracy

Variables

```
cout << "Year:" << 2016;
```

```
int year = 2016;
cout << "Year:" << year;
```

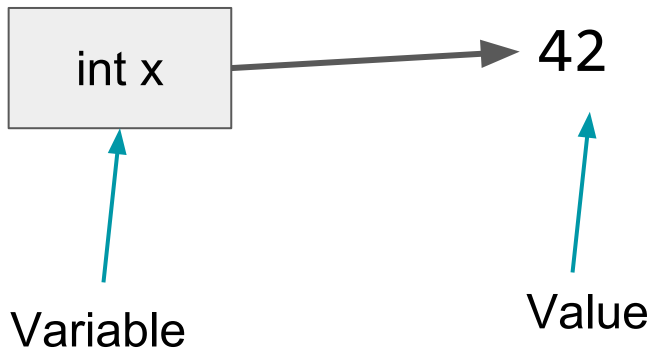
`Output:`
Year: 2016

`Output:`
Year: 2016

Variables allow us to keep track of values by **name**.

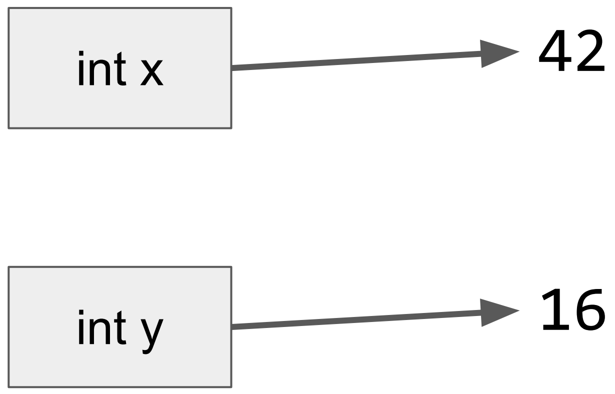
Visualizing Variables

```
int x = 42;
```



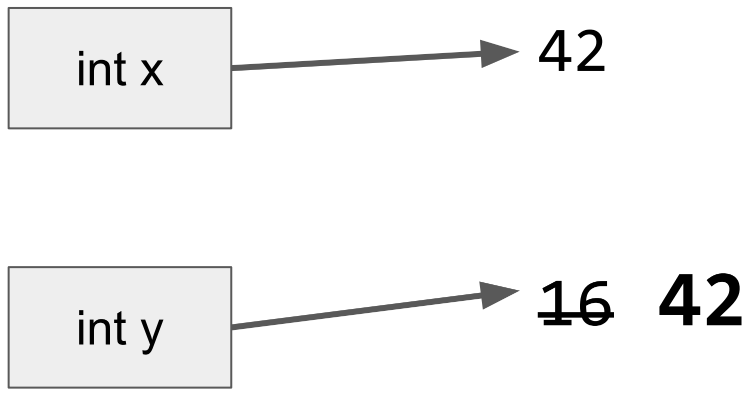
Visualizing Variables

```
int x = 42;
int y = 16;
```



Visualizing Variables

```
int x = 42;
int y = 16;
y = x;
cout << "y:"<< y;
```

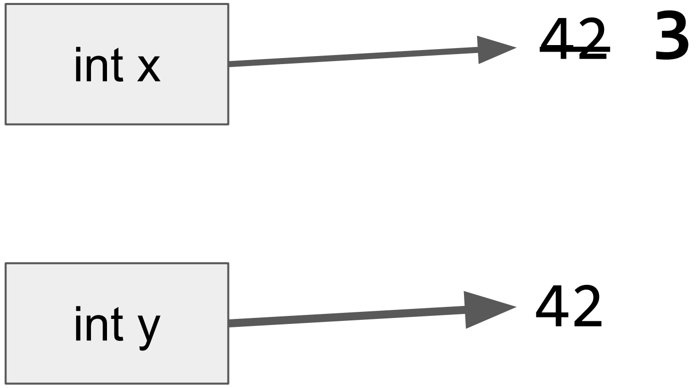


Output:

```
y:42
```

Visualizing Variables

```
int x = 42;
int y = 16;
y = x;
cout << "y:"<< y;
cout << endl;
```
```
x = 3;
cout << "x:"<< x;
cout << endl;
cout << "y:"<< y;
```

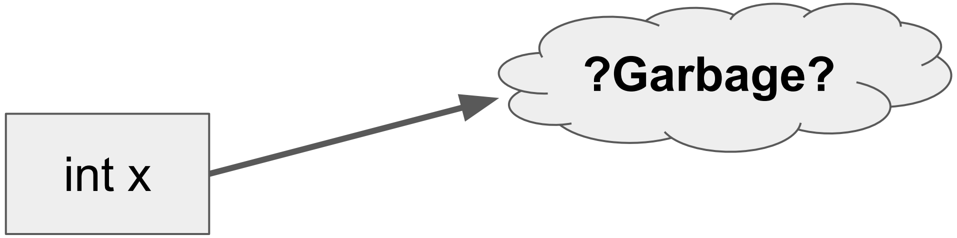


Output:

```
y:42
x:3
y:42
```

Declaring Variables

```
int x;
```



Declares that a variable x of type int exists.

Warning: Since x was not set to any value (initialized), x will point to some "garbage" value. Don't use uninitialized variables!

In Visual Studio 2013, using *uninitialized variables* is a **compilation error**.

Uninitialized Variables

```
int x;
cout << "x is: " << x;
```

This code will **not** compile, because we are trying to use an uninitialized variable.

Initializing Variables

```
int x;
```

Declare variable x

```
x = 42;
```

Initialize variable x to have value 42

```
int x = 42;
```

Declare **and** initialize x

Multiple Declarations

```
int x, y, z;
x = 3;
y = 5;
z = 7;
```

Declare several variables at once

```
double x = 3, y = 1;
```

Declare **and** initialize variables

```
int a, b = 42, c;
a = 1;
c = 8;
```

Can mix and match.

Note: All multiple-declared variables are the **same type**.

Order of Evaluation

```
int x = 2, y = 5;
x = x + y + 1;
```

Question: What is the final value of x?

Answer: 8

```
x = x + y + 1;
=> x = 2 + 5 + 1;
=> x = 8;
```

When evaluating an assignment statement:
(1) Evaluate the right-hand-side (RHS)
(2) Assign the LHS to the RHS's value

Mixing Data Types (int, double)

- Rule of Thumb: When operating on both int's and double's, the resulting value's type is *upgraded* to the **larger/more-expressive** type
  - Example: double can handle more values than int

```
int x = 3;
double y = 4.2;
cout << x + y;
```

$\underbrace{\texttt{x + y}}_{\text{Type was upgraded to double}}$

Question: What is the output?

Answer: 7.2

Data Type Exercises

```
int x = 3;
int y = 4.2;
cout << x + y;
```

4.2 is truncated to 4 when assigning to an int type

Output: 7

```
int x = 3;
double y = 4.2;
double z  =  x + y;
cout << z;
```

Type upgraded to double

Output: 7.2

Data Type Exercises

```
int a = 3;
cout << a/2 << endl;
cout << a/2. << endl;
```

Output:

```
1
1.5
```

a/2 is dividing int by an int. Final type is an **int**. Truncate 1.5 to 1.
Result: a/2 -> 1

a/2. is dividing int by a **double**. Final type is a **double**.
Result: a/2 -> **1.5**

Note: 2. is shorthand for 2.0

Casting (static_cast)

- Can explicitly tell compiler to treat a value as a certain type (ie int or double)

```
int x = 3;
double y = 4.2;
cout << x + y;
```

Type is *implicitly* upgraded to double

Output for Both: 7.2

```
int x = 3;
double y = 4.2;
cout << static_cast<double>(x) + y;
```

Explicitly treat value as a double

static_cast

Syntax: `static_cast<NEWTYPE>(<EXPR>);`

Example:

```
int x = 1;
cout << x / 2 << endl;
cout << static_cast<double>(x) / 2 << endl;
```

Output:

```
0
0.5
```

Exercise: static_cast

```
int x = 2;
cout << static_cast<double>(x / 4) << endl;
cout << static_cast<int>(x / 4.0) << endl;
cout << x / static_cast<double>(4) << endl;
```

Question: What is the output?

Answer:

```
0
0
0.5
```

char

- Used to store single characters
- Use **single quotes** to define char's

```
char c1 = 'E';
char c2 = 'K';
cout << "My initials are: " << c1 << c2;
```

Output:

```
My initials are: EK
```

char: Single vs Double Quotes

- Careful - don't use double-quotes for char's!

```
char c1 = "E";
char c2 = 'K';
cout << "My initials are: " << c1 << c2;
```

Compiler error: complains that you can't assign a char to something in double-quotes.

bool

- Boolean. Data type used to store either *true* or *false*.
- Example:

```
bool mybool1 = true;
bool mybool2 = false;
cout << "mybool1: " << mybool1 << endl;
cout << "mybool2: " << mybool2;
```

Output:

```
mybool1: 1
mybool2: 0
```

Note: Very common for programming languages to treat "true" as 1, and "false" as 0.

We'll likely use bool more when we learn about if statements, for loops, and while loops.

cin: Getting User Input

- Can ask for user input using `cin`: Console Input
  - Defined by <iostream> library (A C++ standard library)
- Example:

```
int myage;
cout << "What is your age?" << endl;
cin >> myage;
cout << "You are " << myage << " years old.";
```

Try it out in Visual Studio!

Chaining cin

- Like cout, one can chain together multiple cin's

```
int x, y;
cin >> x >> y;
```

User can input separate values in *two* different ways:

Option 1: Separate values by *spaces*

```
42 9<ENTER>
```

Option 2: Separate values by *newlines*

```
42<ENTER>9<ENTER>
```

Binary/Decimal

Decimal (Base 10)	Binary (Base 2)
3	0011
2	
	1000
15	
	1001
	0111

Fill in the table, converting to/from decimal/binary as necessary.

[From discussion 2b problems, question 4]

Binary/Decimal

Decimal (Base 10)	Binary (Base 2)
3	0011
2	0010
8	1000
15	1111
9	1001
7	0111

[From discussion 2b problems, question 4]

Operator Shorthands

```
int i = 0;
```

$$\underbrace{\texttt{i++;}\qquad\qquad\texttt{++i;}\qquad\qquad\texttt{i += 1;}}$$

All have the same effect: increment i by 1.

Also equivalent: `i = i + 1;`

Operator shorthands

- In addition to x++, ++x, and x+=3, also have
  - x--, --x, x -= 3
  - x *=3, x /= 3
- Difference between ++x, x++
  - x++: Postfix. Increments x to (x+1), but evaluates to x.
  - ++x: Prefix. Increments x to (x+1), and evaluates to (x+1)
- Example:

```
int x = 2;
cout << x++ << endl;
cout << x << endl;
x = 2;
cout << ++x << endl;
cout << x << endl;
```

Output:

```
2
3
3
3
```

The Mod Operator: %

- AKA remainder
- Syntax: a % b, where a,b are **positive** integers.
- In math notation, we express this as:
  - a mod b

```
int x = 1 % 3;    // x is 1
x = 2 % 3;    // x is 2
x = 3 % 3;    // x is 0
x = 4 % 3;    // x is 1
x = 5 % 3;    // x is 2
x = 6 % 3;    // x is 0
...
```

Trivia: According to C++ spec: if either are negative, then results are "implementation-defined", which means different compilers/machines are allowed to give whatever result they like. Scary.

In Visual Studio 2013, they define modulo as:
x%y is x - y*(x/y)

This handles both positive and negative values of x,y.

Mod

Think of `5 % 3` as: the *remainder* of doing 5/3:

$$5/3 = 1 + (\underbrace{\mathbf{2}}_{}/3)$$

5 / 3 = **1**

Integer division yields the **quotient**.

5 % 3 = **2**

The **remainder**

Numerical Errors

- Overflow: When the value of an int/double exceeds the maximum value
    - Example: Recall that an int has a range of about -2 billion to +2 billion.

```
int x = 2e9; // 2 billion
cout << "x=" << x << endl;
cout << "2*x=" << 2*x << endl;
```

Output:

```
x=2000000000
2*x=-294967296
```

$$2*x=\underbrace{-294967296}_{\text{Woah, is negative?!}}$$

Numerical Errors

- **Underflow** is when a value is smaller than the data type's smallest value
- Precision errors
  - Recall: double has roughly 15 digits of precision

```
double a = 2;
double b = sqrt(a)*sqrt(a) - 2;
cout << "sqrt(2)*sqrt(2) - 2 = " << b << endl;
```

Output:

$$\texttt{sqrt(2)*sqrt(2) - 2 = }\underbrace{\texttt{4.44089e-016}}_{\text{Woah, not exactly 0!}}$$

cin: Input buffering

```
int x;
cin >> x;
```

What cin does:
(1) **Skip** all whitespace (spaces, tabs, newlines) until it reaches a non-whitespace character.
(2) Attempts to interpret the current character as the desired type (int, double, string, etc.).
    If <u>success</u>: **chomp** the character, and move onto the next character. Stops as soon as we either find whitespace, or an inappropriate character. Internal buffer pointer remains at the 'blocking' character (ie \n).
    If <u>fail</u>: cin enters a **failure state**, and will cease to work until it is reset to a normal state: cin.clear().

cin: Example

```
int x;
double y;
cin >> x;
cin >> y;
```

Suppose the user typed:

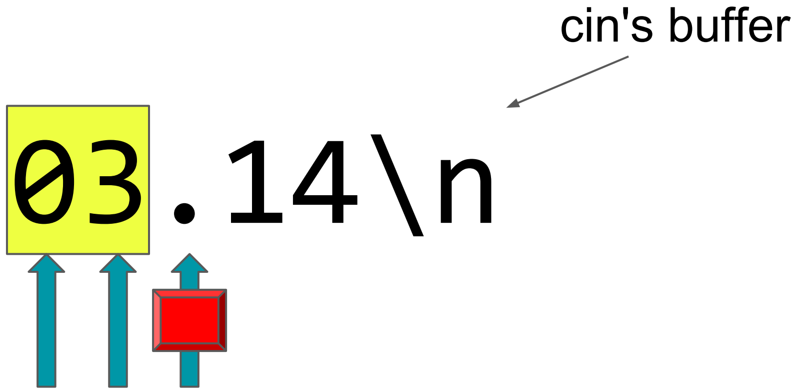
```
03.14<ENTER>
```

What're the values of x and y?

Answer: x is 3, and y is 0.14

cin: Step by Step

```
int x;
double y;
cin >> x;
cin >> y;
```



'.' is not valid for type `int`!

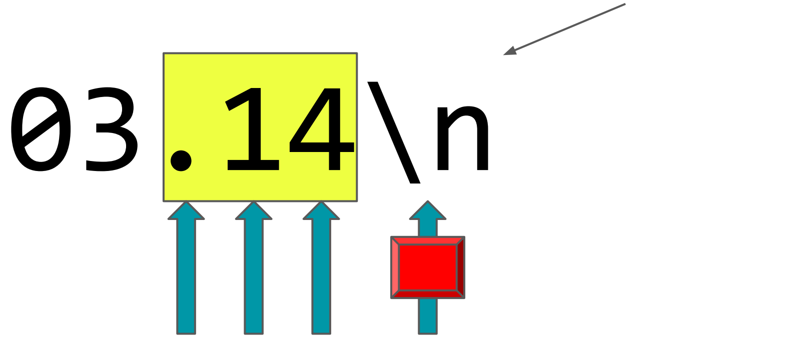
User typed:
`03.14<ENTER>`

Outcome:
cin sets x to: 3

Note: cin remembers its position in the buffer for next time.

cin: Step by Step

```
int x;
double y;
cin >> x;
```
`cin >> y;`



Stop: reached a whitespace character (newline).

User typed:
```
03.14<ENTER>
```

Outcome:
cin sets y to: 0.14

Note: cin's internal buffer pointer still points to the \n character.

cin: Handling Errors

```
int x, y;
cin >> x;
cout << "x is: " << x << endl;;
cin >> y;
cout << "y is: " << y;
```

Suppose user types:

```
d3<ENTER>
```

Output:

```
x is: -858993460
y is: -858993460
```

Uhoh! x, y not set.
Note: we are using x **without initializing** it with a value, hence why this value is so strange.

cin: Handling Errors

cin's buffer

```
int x; char y;
cin >> x;
cin >> y;
```

```
d3\n
```

cin status: **FAILURE**.

cin sees that 'd' is **invalid** for type `int`.

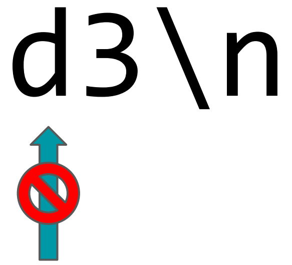
User typed:
`d3<ENTER>`

Outcome:
cin enters a <u>failure state</u>, and "passes out".
cin does not set x to any value.
Any further attempts to use cin will **not do anything**!

cin: Handling Errors

```
int x; char y;
cin >> x;
cin >> y;
```

cin's buffer



cin status: **FAILURE.**

cin is in a failure state, so **does nothing**.

User typed:
`d3<ENTER>`

Outcome:
cin does not set y to any value.

cin: How to fix failure state?

`cin.clear()` is a function that resets cin's state from "Failure" to "Good".

Use it to wake up a "passed out" cin.

cin.clear(): Example

```
int x = 8;
double y;
cin >> x;
cin.clear();
cin >> y;
cout << "x: " << x << endl;
cout << "y: " << y;
```

User types:

```
.45<ENTER>
```

Question: What is the output?

Output:

```
x: 8
y: 0.45
```

cin and strings

- Recall: it's fairly annoying to get more than one word into a string with cin:

```
string name;
cin >> name;
cout << "Name: " << name;
```

User inputs:
Louis Reasoner<ENTER>

Output:

```
Name: Louis
```

cin only set name to first word ("Louis"), since cin stops at first whitespace!

getline

- To get an entire line from the user into a string, use the getline function

```
string name;
getline(cin, name);
cout << "Name: " << name;
```

User inputs:

```
Louis Reasoner<ENTER>
```

Output:

```
Name: Louis Reasoner
```

getline takes everything the user types up to <ENTER>, and puts it into the string "name".