PIC 10A 1A

TA: Bumsu Kim

UCLA

Today…

- How to use PIC LAB
- Visual Studio Shortcuts
- Arithmetic Operations on `int` and `double`
- Homework Tips
- Exercise

PIC LAB

- PIC LAB is on the second floor of this building (2000 Math Science Building)
- There are two rooms (Left/Right) and my office hours will be held in the left room
- You will need a PIC account to log in

How to log in to your PIC account

Your PIC account was automatically created for you within 24 hours of your registration for this course. To log into your Windows account for the first time, enter your username and password. Your username should be the same as your Bruin online name, or your first initial / last name (e.g. jbruin). Your password is initially your 9-digit student ID number. After logging in, you will be ask to create a new password. It has to be at least 8 characters long, with both numbers and letters.

UCLA

Visual Studio Shortcuts

- After selecting a part of your code,
- [Ctrl] + K + F : Auto indentation/Spacing
- [Ctrl] + K + C : Make Comment
- [Ctrl] + K + U : Uncomment

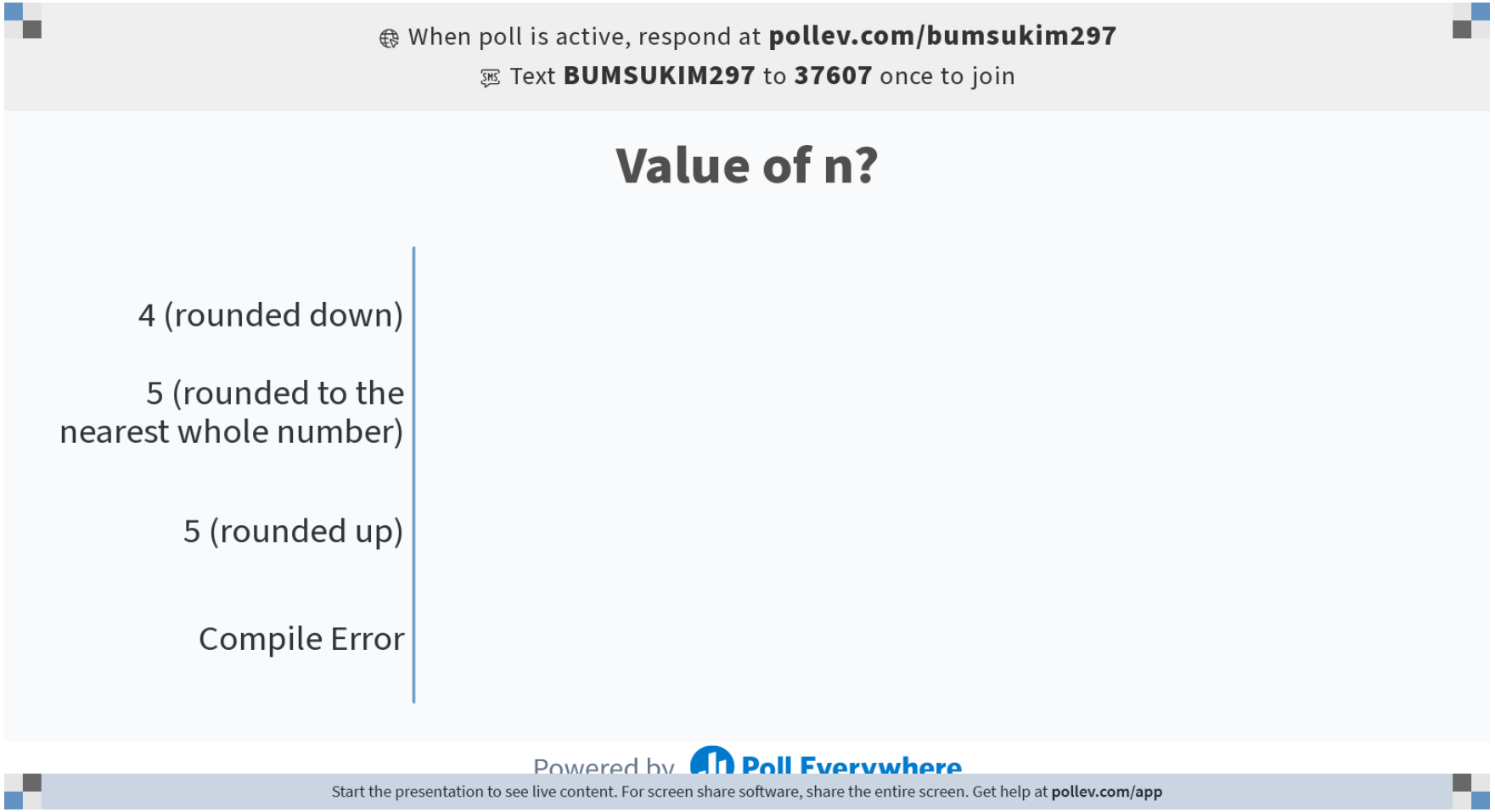
- To run your code,
- [Ctrl] + [F5] : Run without debugging (this should be used by default)
- [F5] : Start debugging (when things went wrong, you can use this)

Integer Operations (Review)

- Recall that **+**, **-**, and * work as we expect from everyday math
- However, **/** for two integers does “integer division”
  - We only take the quotient, and the remainder is discarded
- On the other hand, **%** finds the remainder
- For example, 14/3 == 4 and 14%3 == 2, because 14 = 3*4 + 2
- Maybe a similar question:

```
int n = 4.567; // what is the value of n?
```

```
int n = 4.567; // what is the value of n?
```



More on `int` operators

- Special operators for integral types (for now):
  - ++, ++, --, and --
  - (Post/pre)-(increment/decrement) operators

```
int n = 3;
++n; // pre-increment
n++; // post-increment
--n; // pre-decrement
n--; // post-decrement
```

- Pre and Post operators are different (in terms of their returned values)
- In general, pre-increment/decrement is preferred because, internally, there's a non-necessary copy in post-operators

More on operators

- Compound assignments: +=, -=, *=, /=

- Usage: x = x [operator] y;
  - [operator] can be +, *, -, / (and others)

```
int n = 3;
n += 2; // n = n + 2
n -= 2; // n = n - 2
n *= 2; // n = n * 2
n /= 2; // n = n / 2
```

- Q: What happens if you do `n %= 2` when `n` is even/odd (respectively)?

Floating-point Numbers

- Last time, I mentioned that the type determines how the “bits” should be read and interpreted
- For instance, we have different bits for x and y here:

```
double x = 5;
int y = 5;
```

- Let’s talk more about floating point numbers

Floating-point Numbers

- Floating-point types: uses the "scientific notation," in base 2 numbers
  - Ex) 1902849287.4356 → 1.902×10^{9}
  - Ex) 0.0000034453234 → 3.445×10^{-6}
  - Loses precision, but more efficient for finite storage

- $\text{Mantissa} \times \text{Base}^{\text{Exponent}}$

- Of course, the Base is 2, and the mantissa and the exponent are represented in base 2 numbers

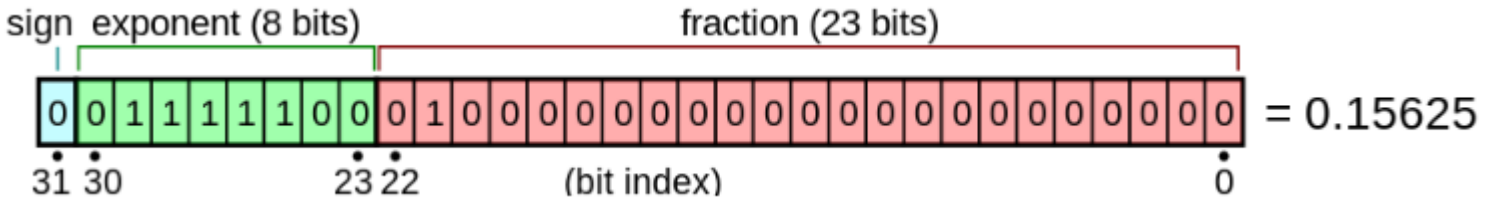
- For instance, "4.35" is stored approximately as 4.34999999999999645
  - Example:

```
int n = 4.35 * 100;
```

UCLA

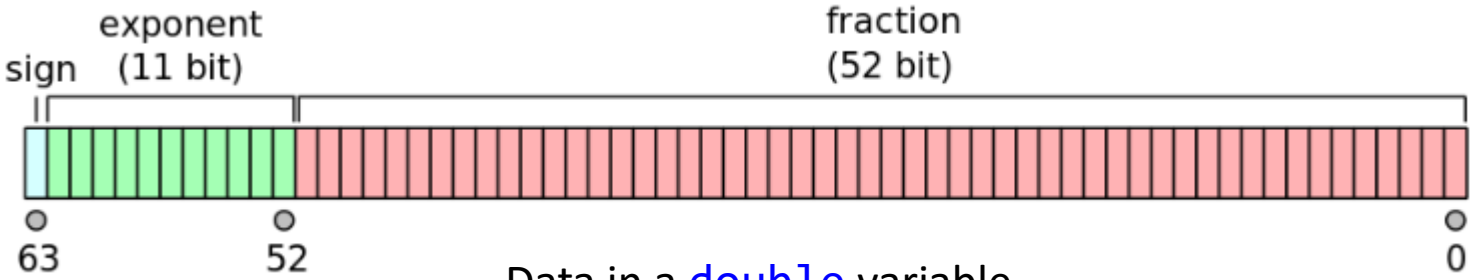
Floating-point Numbers

- Floating-point types:
  - `float`: 4 bytes
  - `double`: 8 bytes



Data in a `float` variable

- Double means “double-precision”



Data in a `double` variable

- 23-bit mantissa → about 7 significant digits in decimal (`float`)
- 52-bit mantissa → about 15 significant digits in decimal (`double`)

UCLA

The type `double` – operators

- Operators such as +, -, *, / are defined for the type `double`
- Fortunately, they are not as tricky as the case of `int`
- Use them like real number arithmetic operators, just don't forget that they have finite precision, and upper/lower limits
- There is a small number ***eps*** such that **1 + *eps* == 1** !!
  - Called a machine epsilon, and sets the limit of precision in your number system
  - Example in Live Coding

Implicit and Explicit Casts

- Always use EXPLICIT casts

```
double x = 1;  // bad: implicit casting
int y = 1.5;   // worse: loss of information
double y = 1.0; // good
int z = static_cast<int>(1.5); // good
```

- Use `static_cast` when you should make it clear

- You will see more confusing implicit typecasts later
- So let's abide by a good coding practice

Participation Question

- Suppose that `runTime` is a double variable storing the number of seconds (down to the nearest hundredth of a second) it takes someone to complete a marathon.

- Consider the code below that is to compute their time, rounded to the nearest second, in the format of hours, minutes, and seconds (seconds and minutes should not exceed 60).
  Does the code work?


```
int a = static_cast<int>(runTime + 0.5);
int hours = a / 3600;
int minutes = (a % 3600) / 60;
int seconds = a % 60;

cout << "Rounded run time: "
       << hours << " hour(s), " << minutes
       << " minute(s), " << seconds << " second(s)";
```

A) No, there is a logical error and the result is wrong.
B) Yes, but the coding style needs improvement
C) Yes, and the style is fine.