Numeric Data Types Outline

1. Numeric Data Types Outline
2. Data Types
3. Integers in Mathematics
4. Integers in Computing
5. Integers A.K.A. Fixed Point Numbers
6. Declaring `int` Variables
7. `int` Data Don’t Have to Be 4 Bytes Long
8. `int` Declaration Example Program Part 1
9. `int` Declaration Example Program Part 2
10. The Same Source Code without Comments
11. `int` Literal Constants
12. `int` Literal Constants Usage
13. `int` Literal Constants Usage: Good & Bad
14. `int` Named Constants Example #1
15. `int` Named Constants Example #2
16. Real Numbers in Mathematics
17. Reals: Digits to the Right of the Decimal
18. A Fun Rational Number
19. Integers vs Reals in Mathematics
20. Representing Real Numbers in a Computer
21. `float` Literal Constants
22. Declaring `float` Variables
23. `float` Variable Size
24. `float` Declaration Example Part 1
25. `float` Declaration Example Part 2
26. The Same Source Code without Comments
27. Scientific Notation
28. Floating Point Numbers
29. `float` Approximation #1
30. `float` Approximation #2
31. `float` Approximation #3
32. `float` Approximation Example Program
33. Floating Point Approximation Examples
34. `float` Literal Constants
35. `float` Literal Constant Examples
36. `float` Literal Constants Usage
37. `float` Lit Constant Usage: Good & Bad
38. `float` Named Constants Example Program #1
39. `float` Named Constants Example Program #2
40. Why Have Both Reals & Integers? #1
41. Why Have Both Reals & Integers? #2
42. Programming Exercise

Data Types

A **data type** is (surprise!) a type of data:

- Numeric
  - `int`: ***integer***
  - `float`: ***floating point*** (also known as ***real***)
- Non-numeric
  - `char`: ***character***

Note that this list of data types **ISN'T** exhaustive – there are many more data types (and you can define your own).

```
#include <stdio.h>
int main ()
{ /* main */
    float standard_deviation, relative_humidity;
    int   count, number_of_silly_people;
    char  middle_initial, hometown[30];
} /* main */
```

Integers in Mathematics

Mathematically, an ***integer*** is:

any number (positive, negative or zero)

that has nothing but zeros to the right of its decimal point:

-3984.00000000...

0.00000000...

23085.00000000...

Another way to think of integers is as

- the counting numbers, and

  1, 2, 3, 4, 5, 6, ...
- their negatives (additive inverses), and

  -1, -2, -3, -4, -5, -6, …
- zero.

In **mathematics**, the range on integers is infinite in both directions:

$-\infty$ to $+\infty$

Integers in Computing

An integer in **computing** has mostly the same mathematical properties as an integer in mathematics.

An integer in **computing** has a finite range (minimum, maximum).

An integer in computing also has a particular way of being represented in memory (which we'll see later in the course, time permitting) and a particular way of being operated on.

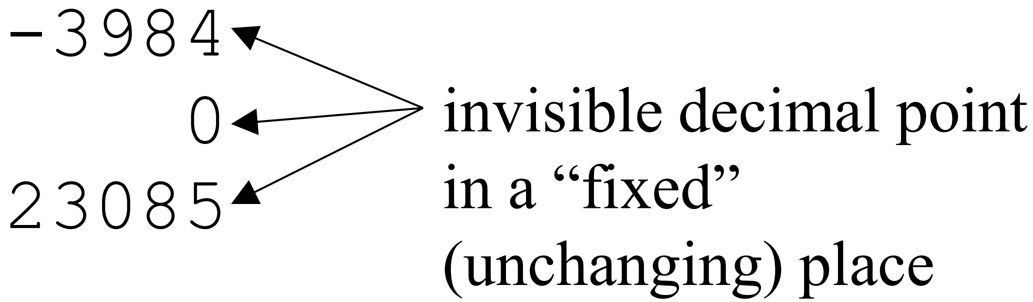
In C (and in most programming languages), `int` literal constants are expressed **without a decimal point**:

```
-3984
    0
23085
```

Integers A.K.A. Fixed Point Numbers

Integers are also known as ***fixed point*** numbers, because they have an invisible decimal point in a **fixed** (unchanging) position.

Specifically, every integer’s invisible decimal point is to the right of the rightmost digit (the “ones” digit):



```
-3984
    0
23085
```

invisible decimal point in a “fixed” (unchanging) place

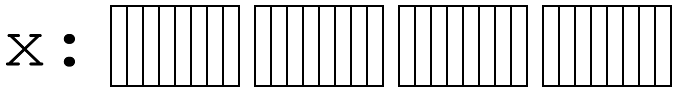
Declaring `int` Variables

```
int x;
```

This declaration tells the compiler to grab a group of bytes, name them `x`, and think of them as storing an `int`.

How many bytes?

That depends on the platform and the compiler, but these days the **<u>typical</u>** answer is that an `int` takes 4 bytes (32 bits) in most cases:



For example, on x86-based Linux PCs such as `ssh.ou.edu`, using the `gcc` compiler from `gnu.org` (the compiler that we're using in this course), the size of an `int` is 4 bytes.

`int` Data Don’t Have to Be 4 Bytes Long

On some ***platforms*** (combination of hardware family and operating system), on some compilers, **all** `ints` are 4 bytes.

On other platforms, the **default** `int` size is 4 bytes, but the size of an `int` can be changed by using a compiler option.

Notice that different compilers for the same language can have different names, different defaults and different options.

While there are many common features, compiler vendors are under no compulsion to follow them.

int Declaration Example Program Part 1

```
% cat assign.c
/*
 *********************************************
 *** Program: assign                       ***
 *** Author: Henry Neeman (hneeman@ou.edu) ***
 *** Course: CS 1313 010 Spring 2025       ***
 *** Lab: Sec 012 Fridays 1:00pm           ***
 *** Description: Declares, assigns and    ***
 *** outputs a variable.                   ***
 *********************************************
 */
#include <stdio.h>

int main ()
{ /* main */
   /*
    *
    ******************************************
    * Declaration section                    *
    ******************************************
    *
    *******************
    * Local variables *
    *******************
    *
    * height_in_cm: my height in cm
    */
    int height_in_cm;
```

int Declaration Example Program Part 2

```
   /*
    *********************************************
    * Execution section *
    *********************************************
    * Assign the integer value 160 to height_in_cm.
    */
    height_in_cm = 160;
   /*
    * Print height_in_cm to standard output.
    */
    printf("My height is %d cm.\n", height_in_cm);
} /* main */
% gcc -o assign assign.c
% assign
My height is 160 cm.
```

The Same Source Code without Comments

```
% cat assign.c
#include <stdio.h>

int main ()
{ /* main */
    int height_in_cm;

    height_in_cm = 160;
    printf("My height is %d cm.\n", height_in_cm);
} /* main */
% gcc -o assign assign.c
% assign
My height is 160 cm.
```

`int` Literal Constants

An ***<u>`int` literal constant</u>*** is any sequence of digits, possibly preceded by an optional sign:

CORRECT: `0      -345     768      +12345`

INCORRECT:

- `1,234,567`
  INCORRECT: <u>No commas</u> allowed.
- `12.0`
  INCORRECT: <u>No decimal point</u> allowed.
- `--4      ++3`
  INCORRECT: A maximum of **<u>one sign</u>** per int literal constant.
- `5-       7+`
  INCORRECT: The sign must come **<u>before</u>** the digit(s), not after.

`int` Literal Constants Usage

We can use `int` literal constants in several ways:

- In declaring and initializing a **named constant**:
```
const int w = 0;
/* 0 is an int literal constant */
```
- In **initializing** a variable (within a declaration):
```
int x = -19;
/* -19 is an int literal constant */
```
- In an **assignment**:
```
y = +7;
/* +7 is an int literal constant */
```
- In an **expression** (which we'll learn more about):
```
z = y + 9;
/* 9 is an int literal constant */
```

`int` Literal Constants Usage: Good & Bad

We can use `int` literal constants in several ways:

- In declaring and initializing a **named constant**:

```
const int w = 0;
/* This is GOOD. */
```

- In **initializing** a variable (within a declaration):

```
int x = -19;
/* This is GOOD. */
```

- In an **assignment**:

```
y = +7;
/* This is BAD BAD BAD! */
```

- In an **expression** (which we'll learn more about):

```
z = y + 9;
/* This is BAD BAD BAD! */
```

`int` Named Constants Example #1

```
#include <stdio.h>
int main ()
{ /* main */
    const int number_of_people_to_tango =   2;
    const int inches_per_foot           =  12;
    const int degrees_in_a_circle       = 360;

    printf("It takes %d to tango.\n",
        number_of_people_to_tango);
    printf("\n");
    printf("There are %d inches in a foot.\n",
        inches_per_foot);
    printf("\n");
    printf("There are %d degrees in a circle.\n",
        degrees_in_a_circle);
} /* main */
```

int Named Constants Example #2

```
% gcc -o intconsts intconsts.c
% intconsts
It takes 2 to tango.

There are 12 inches in a foot.

There are 360 degrees in a circle.
```

ASIDE: Notice that you can output a blank line by outputting a string literal containing only the newline character `\n`.

Real Numbers in Mathematics

Mathematically, a ***real number*** is
a number (positive, negative or zero) with
any sequence of digits on either side of the decimal point:

-3984.75

0.1111111...

3.1415926...

In **mathematics**, the range on real numbers is
infinite in both directions:

-∞ to +∞

Reals: Digits to the Right of the Decimal

In **mathematics**, the string of digits to the right of the decimal point can be either:

- ***terminating*** (a finite number of nonzero digits, maybe even **NO** nonzero digits), **OR**
- ***repeating*** (a finite sequence of digits repeated infinitely), **OR**
- ***non-repeating***.

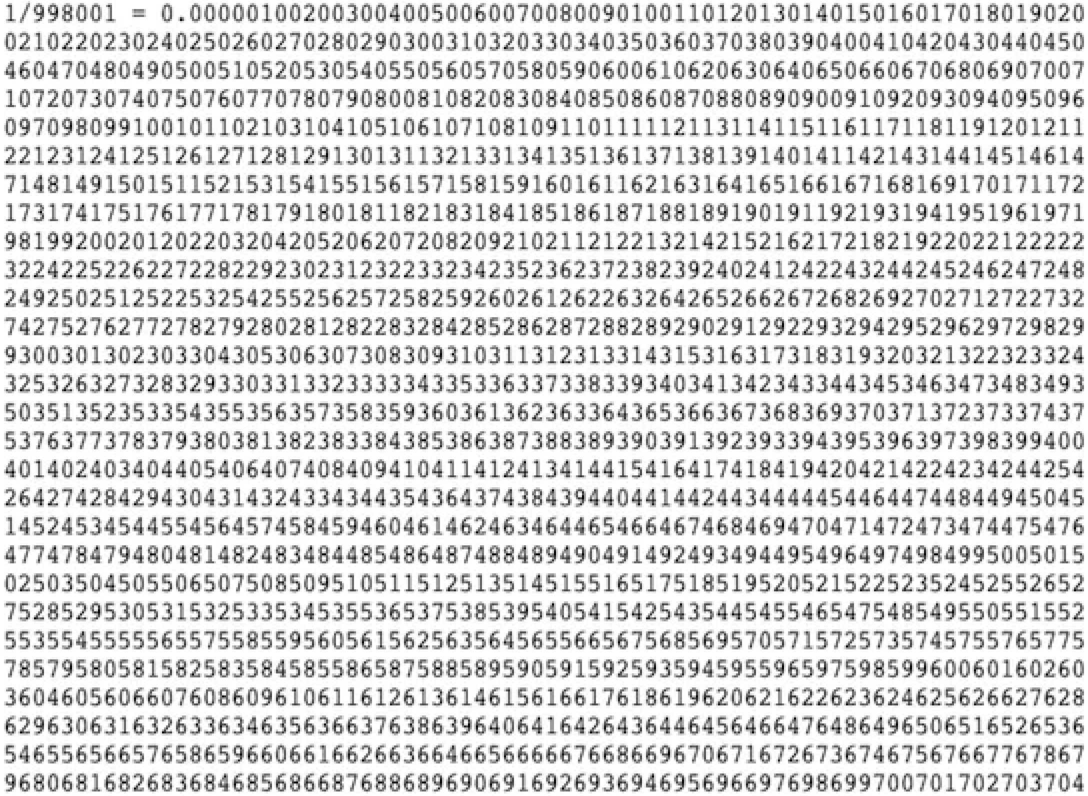
In **mathematics**, there are infinitely many real numbers.

In fact, there are infinitely many real numbers between any two real numbers.

For example, there are infinitely many real numbers between 0 and 0.0000000000000001.

A Fun Rational Number

1 / 998,001 has, as its repeating decimal expansion, every 3-digit integer from 000 to 999, in order, **EXCEPT** 998:



```
1/998001 = 0.000001002003004005006007008009010011012013014015016017018019020
021022023024025026027028029030031032033034035036037038039040041042043044045O
```

https://img-9gag-fun.9cache.com/photo/aK69rpN_700bwp.webp

Integers vs Reals in Mathematics

Notice that, in **mathematics**, all integers are real numbers, but not all real numbers are integers.

In particular, **mathematically**, every integer is a real number, because it has a finite number of nonzero digits to the right of the decimal point.

Specifically, an integer has **<u>NO</u>** nonzero digits to the right of the decimal point.

Representing Real Numbers in a Computer

In a **computer**, a real value is stored in a finite number of bits (typically 32 or 64 bits).

But there are infinitely many real numbers, and in fact there are infinitely many real numbers between any two real numbers.

So a computer's representation of real numbers can only <u>**approximate**</u> most mathematical real numbers.

This is because only finitely many different values can be stored in a finite number of bits.

For example, 32 bits can have only 2^{32} possible different values.

A real value in computing has a finite range (minimum, maximum).

Like integers, real numbers have particular ways of being represented in memory and of being operated on.

`float` Literal Constants

In C (and in most programming languages), `float` literal constants often are expressed **with** a decimal point:

```
-3984.75
     0.0
23085.1235
```

Recall that, in mathematics, all integers are reals, but not all reals are integers.

Similarly, in most programming languages, some real numbers are mathematical integers (for example, 0.0), even though they are represented in memory as reals.

In computing, reals are often called ***floating point*** numbers. We'll see why soon.

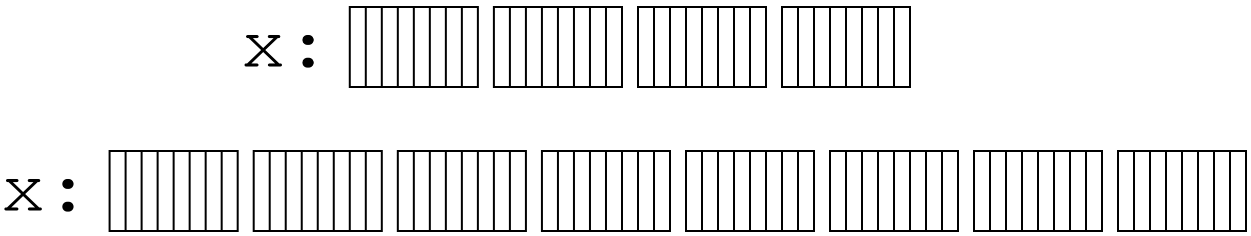
Declaring `float` Variables

```
float x;
```

This declaration tells the compiler to grab a group of bytes, name them `x`, and think of them as storing a `float`, which is to say a real number.

How many bytes?

That depends on the platform and the compiler, but these days the **typical** answer is that real numbers in most cases take 4 bytes (32 bits) or 8 bytes (64 bits):



`float` Variable Size

For example, on x86-based Linux PCs such as `ssh.ou.edu`, using the `gcc` compiler from `gnu.org`, which we're using in this course, the default size of a `float` is 4 bytes (32 bits).

`float` Declaration Example Part 1

```
% cat realassign.c
/*
 *********************************************
 *** Program: realassign                   ***
 *** Author: Henry Neeman (hneeman@ou.edu) ***
 *** Course: CS 1313 010 Spring 2025       ***
 *** Lab: Sec 012 Fridays 1:00pm           ***
 *** Description: Declares, assigns and    ***
 ***   outputs a real variable.            ***
 *********************************************
 */
#include <stdio.h>

int main ()
{ /* main */
   /*
    *
    ******************************************
    * Declaration section                    *
    ******************************************
    *
    *******************
    * Local variables *
    *******************
    *
    * height_in_m: my height in m
    */
    float height_in_m;
```

float Declaration Example Part 2

```
   /*
    *********************************************
    * Execution section *
    *********************************************
    * Assign the real value 1.6 to height_in_m.
    */
    height_in_m = 1.6;
   /*
    * Print height_in_m to standard output.
    */
    printf("My height is %f m.\n", height_in_m);
} /* main */
% gcc -o realassign realassign.c
% realassign
My height is 1.600000 m.
```

The Same Source Code without Comments

```
% cat realassign.c
#include <stdio.h>

int main ()
{ /* main */
    float height_in_m;

    height_in_m = 1.6;
    printf("My height is %f m.\n", height_in_m);
} /* main */
% gcc -o realassign realassign.c
% realassign
My height is 1.600000 m.
```

Scientific Notation

In technical situations, we often encounter ***scientific notation***, which is a way of writing numbers that are either **very very big** or **very very small**:

6,300,000,000,000,000 = 6.3×10^{15}

0.0000000000271 = 2.71×10^{-11}

In C, we can express such numbers in a similar way:

6,300,000,000,000,000 = `6.3e+15`

0.0000000000271 = `2.71e-11`

Here, the `e`, which is short for "exponent," indicates that the sequence of characters to the right of the `e` – an optional sign followed by one or more digits – is the power of 10 that the number to the left of the `e` should be multiplied by.

Floating Point Numbers

When we express a real number in scientific notation, the decimal point is immediately to the right of the leftmost non-zero digit.

So, the decimal point doesn't have to be to the right of the "ones" digit; instead, it can be after **any** digit. It doesn't have a fixed location, so we say that it ***floats***.

So, we sometimes call real numbers ***floating point*** numbers.

We recall that, similarly, integers are sometimes called ***fixed point*** numbers, because they have an implicit decimal point that is in a fixed location, always to the right of the "ones" digit (that is, the rightmost digit), with implied zeros to the right of the implied decimal point:

6, 300, 000, 000, 000, 000 = 6, 300, 000, 000, 000, 000.0000 . . .

`float` Approximation #1

In C (and in most other programming languages), real numbers are represented by a finite number of bits.

For example, on Linux PCs like `ssh.ou.edu`, the default size of a `float` is 32 bits (4 bytes).

We know that 32 bits can store

$$2^{32} = 2^{2} \times 2^{30} = 2^{2} \times 2^{10} \times 2^{10} \times 2^{10}$$
$$\simeq 4 \times 10^{3} \times 10^{3} \times 10^{3} =$$

roughly 4,000,000,000 possible values.

That’s a lot of possibilities.

<u>But</u>: There are infinitely many (mathematically) real numbers, and in fact infinitely many real numbers between any two real numbers.

`float` Approximation #2

For example, between 1 and 10 we have:

2	3	4	5	6	7	8	9
2.9	3.8	4.7	5.6	6.5	7.4	8.3	9.2
2.09	3.08	4.07	5.06	6.05	7.04	8.03	9.02
2.009	3.008	4.007	5.006	6.005	7.004	8.003	9.002
2.0009	3.0008	4.0007	5.0006	6.0005	7.0004	8.0003	9.0002

...

So, no matter how many bits we use to represent a real number, we won't be able to exactly represent most real numbers, because we have an infinite set of real numbers to be represented in a finite number of bits.

`float` Approximation #3

No matter how many bits we use to represent a real number, we won’t be able to exactly represent most real numbers, because we have an infinite set of real numbers to be represented in a finite number of bits.

For example:

if we can exactly represent 0.125 but not

0.12500000000000000000000000000000001,

then we have to use 0.125 to **approximate**

0.12500000000000000000000000000000001.

float Approximation Example Program

```
% cat real_approx.c
#include <stdio.h>

int main ()
{ /* main */
    float input_value;

    printf("What real value would you like stored?\n");
    scanf("%f", &input_value);
    printf("That real value is stored as %f.\n",
        input_value);
} /* main */
% gcc -o real_approx real_approx.c
% real_approx
What real value would you like stored?
0.12500000000000000000000000000001
That real value is stored as 0.125000.
```

Floating Point Approximation Examples

$1.25 = 2^{0} + 2^{-2}$

$0.1 \simeq$

$2^{-4} + 2^{-5} + 2^{-8} + 2^{-9} + 2^{-12} + 2^{-13} + 2^{-16} + 2^{-17} + 2^{-20} + 2^{-21} +$

$2^{-24} + 2^{-25} + 2^{-28} + 2^{-29} + 2^{-32} + 2^{-33} + 2^{-36} + 2^{-37} + 2^{-40} + 2^{-41} +$

$2^{-44} + 2^{-45} + 2^{-48} + 2^{-49} + 2^{-52} + 2^{-53} + 2^{-55}$

http://bartaz.github.io/ieee754-visualization/

`float` Literal Constants

A ***`float` literal constant*** is:

- an optional sign, followed by
- a sequence of one or more digits (which is optional if there are digits to the right of the decimal point), followed by
- a decimal point (which is optional if there is an exponent), followed by
- an optional sequence of one or more digits to the right of the decimal point (if there is one), followed by
- an optional exponent string, which consists of an `e`, an optional sign, and a sequence of one or more digits.

You can tell that a numeric literal constant is a **`float` literal constant** because it has either a **decimal point**, or an **`e`**, or **both**.

`float` Literal Constant Examples

```
0.0
-345.3847
7.68e+05
+12345.434e-13
125.e1
1e1
```

`float` Literal Constants Usage

We can use `float` literal constants in several ways:

- In declaring and initializing a **named constant**:

  ```
  const float w = 0.0;
  /* 0.0 is a float literal constant */
  ```

- In **initializing** a variable (within a declaration):

  ```
  float x = -1e-05;
  /* -1e-05 is a float literal constant */
  ```

- In an **assignment**:

  ```
  y = +7.24690120;
  /* +7.24690120 is a float literal
   * constant */
  ```

- In an **expression** (which we'll learn more about):

  ```
  z = y + 125e3;
  /* 125e3 is a float literal constant */
  ```

`float` Lit Constant Usage: Good & Bad

We can use `float` literal constants in several ways:

- In declaring and initializing a **named constant**:

```
const float w = 0.0;
/* This is GOOD. */
```

- In **initializing** a variable (within a declaration):

```
float x = -1e-05;
/* This is GOOD. */
```

- In an **assignment**:

```
y = +7.24690120;
/* This is BAD BAD BAD! */
```

- In an **expression** (which we'll learn more about):

```
z = y + 125e3;
/* This is BAD BAD BAD! */
```

`float` Named Constants Example Program #1

```
#include <stdio.h>

int main ()
{ /* main */
    const float pi = 3.1415926;
    const float radians_in_a_semicircle = pi;
    const float number_of_days_in_a_solar_year =
                365.242190;
    const float US_inflation_percent_in_1998 = 1.6;

    printf("pi = %f\n", pi);
    printf("\n");
    printf("There are %f radians in a semicircle.\n",
        radians_in_a_semicircle);
    printf("\n");
    printf("There are %f days in a solar year.\n",
        number_of_days_in_a_solar_year);
    printf("\n");
    printf("The US inflation rate in 1998 was %f%%.\n",
        US_inflation_percent_in_1998);
} /* main */
```

float Named Constants Example Program #2

```
% gcc -o real_constants real_constants.c
% real_constants
pi = 3.141593

There are 3.141593 radians in a semicircle.

There are 365.242188 days in a solar year.

The US inflation rate in 1998 was 1.600000%.
```

Again, notice that you can output a blank line by printing a string literal containing only the newline character `\n`.

Reference:

http://scienceworld.wolfram.com/astronomy/LeapYear.html

Why Have Both Reals & Integers? #1

1. **Precision**: `ints` are exact, `floats` are approximate.
2. **Appropriateness**: For some tasks, `ints` fit the properties of the data better. For example:
   a. counting the number of students in a class;
   b. array indexing (which we'll see later).
3. **Readability**: When we declare a variable to be an `int`, we make it obvious to anyone reading our program that the variable will contain only certain values (specifically, only integer values).

Why Have Both Reals & Integers? #2

4. **Enforcement**: When we declare a variable to be an `int`, no one can put a non-`int` into it.
5. **History**: For a long time, operations on `int` data were much quicker than operations on `float` data, so anything that you could do with `int`s, you would. Nowadays, operations on `float`s can be as fast as (or faster than!) operations on `int`s, so speed is no longer an issue.

Programming Exercise

Write a program that inputs, and then outputs,
the user's number of first cousins and height in meters.

The program should do the following:

1. greet the user;
2. prompt the user to input their number of cousins;
3. input their number of cousins;
4. prompt the user to input their height in meters;
5. input their height in meters;
6. output their number of cousins, in a full sentence;
7. output their height in meters, in a full sentence.

Be sure to use **appropriate data types and placeholders**.