Gotchas

In this lesson, we'll point out some common unintentional mistakes that can be made in C.

We'll cover the following

There are a number of classic **gotchas** in C to watch out for when trying to figure out why your program is not running as expected (or not running at all). Check for these first when debugging, they are common.

Integer Math

Like Python, in C when you write mathematical expressions, be sure to **include decimal places for all numbers** (that is, unless you actually want to do integer math). What I mean is this. What do you think the following code will print to the screen?

```
#include <stdio.h>

int main ()
{
  double mass = 2.5;
  double velocity = 4.5;
  double kinetic_energy = (1/2) * mass * velocity * velocity;
  printf("kinetic energy = %.3f\n", kinetic_energy);
  return 0;
}
```

If you guessed this, you are correct!

Why did this happen? It's because the C expression `(1/2)` in line 7 of the code is evaluated as the **integer** `1` divided by the **integer** 2, which is the **integer** `0`. If we want an expression to represent a floating-point value, then we have to **use floating-point syntax for all numbers**:

```c
#include <stdio.h>

int main ()
{
  double mass = 2.5;
  double velocity = 4.5;
  double kinetic_energy = (1.0/2.0) * mass * velocity * velocity;
  printf("kinetic energy = %.3f\n", kinetic_energy);
  return 0;
}
```

As a general rule, I always put **.0** at the end of all numbers in my C code.

Floating-point Math

Computers use bits (0s and 1s) to represent data, including floating-point numbers. This means numbers that we typically represent using base-10 must be represented in base-2 in the computer.

There are numbers we can express in one way (like the fraction 1/3) that we cannot express precisely in base-10 floating-point. We would have to write `1.333333` and keep going ad infinitum with the `.3333`. Similarly, there are numbers that we can express easily in base-10 that cannot be precisely represented in base-2 (one gets infinitely repeating sequences).

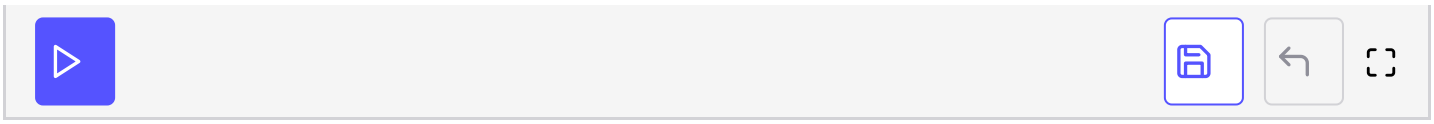
Why am I telling you this? The reason is, that when using floating-point representations like `float` and `double` in C (and the same is true in other languages), there is always some amount of **rounding error** involve in the representation of numbers. There are also a host of other issues one encounters when doing so-called **floating-point arithmetic**, there is an excellent summary of all the issues here: What Every Computer Scientist Should Know About Floating-Point Arithmetic. There is a slightly less intimidating explanation (in the context of Python) here: Floating Point Arithmetic: Issues and Limitations.

The bottom line is, one must always be aware of the limitations of floating-point representation, and precision issues like rounding error. One example: *never use boolean operators like `==` or `<` or `>` or `!=` to test for equality, etc, of floating-point variables or expressions. Here is an example of how you might be led astray. What is the output of this program?

is the output of this program?

```
#include <stdio.h>

int main () {
  double x = 0.1;
  double y = 0.2;
  if ((x + y) == 0.3) printf("TRUE\n"); else printf("FALSE\n");
  printf("x + y = %.20f\n", x+y);
  return 0;
}
```



If you guessed `FALSE` then you are correct.

You can see the rounding error here.

One option is when you need to compare two values to see if they are “equal”, instead check to see if their difference is below some minimum threshold which, you decide in advance, is “equal enough”. If you are dealing with small expected differences in your work however, this may not be a suitable solution.

Another option is to use a library for arbitrary-precision math such as GMP that allows for arbitrary precision representation of numbers. The drawback however is relatively slow speed.

Another web page discussing a floating-point test suite: here

The C99 standard introduced support for the IEEE 754 floating-point standard. The IEEE 754 standard specifies things like arithmetic formats (binary and decimal floating-point data), rounding rules, arithmetic operations and exception handling. If floating-point math is a concern in your code, you should look into compiling your code with the `-std=c99` flag.

Here is another description of floating-point roundoff error, and some real-world examples involving a stock market, a rocket, and a missile: Roundoff Error.

Since we’re on the topic of writing code correctly, we’ll have to touch upon the age-old debate of whether code should be understandable or concise. Let’s do that in the next lesson.