■ 補足：ゼロによる除算

言語を問わず、プログラミングではゼロによる除算を行わないように注意が必要です。データ型だけでなく、処理系によっても振る舞いが異なるものですから、基礎的な理解を得ておくと良いです。

注意：Intel x86 アーキテクチャの CPU でなく ARM アーキテクチャの M1/M2 CPU を使った Mac では挙動が異なります。後半に補記しておきます。

□ 整数型におけるゼロ除算

右に整数型の変数の値が 0 となるコードと、その実行時のメッセージを示します。
出力をよく見ると、“Floating exception”（浮動小数点処理における例外の発生）と表示されているのは printf() 関数がそのように出力したわけでは無く、処理系が出している上に、その時点で処理を中断したことが分かるでしょう。
（ a の printf() は正しく 0 を出力していますが、次の行の出力は b の結果（数字）ではなく文字（Floating…）であり、その次に出るはずの c の結果が一行もありません。つまり処理は中断されたわけです。）

コード：
```
int a, b, c;
a=0;
printf("%d\n", a);
b=10 / a;
printf("%d\n", b);
c=b * 10;
printf("%d\n", c);
```

実行時：
```
$ ./a.out
0
Floating exception
$
```

つまり整数型でのゼロ除算はこのように実行中断を伴う、実行時エラーとなって現れます。

□ 実数型におけるゼロ除算

今度は実数の変数の値が 0.0 となるコードと、その実行時のメッセージを示します。
今度は停止を伴うような実行時エラーは発生せず、printf() が特殊な結果表示を行っています。（数字ですらなく、無限大 (infinity) を意味する inf と文字が出ています。）

厄介なのは「実行時にそのようなことが起きていると気づかない可能性がある」ことと、結果が連鎖的に伝播してしまうことです。右の例で inf を含む演算の結果が再び inf になっていることがわかるでしょう。

コード：
```
double a, b, c;
a=0.0;
printf("%f\n", a);
b=10.0 / a;
printf("%f\n", b);
c=b * 10.0;
printf("%f\n", c);
```

実行時：
```
$ ./a.out
0.000000
inf
inf
$
```

□ 規格

浮動小数点処理におけるこうした特殊な操作は IEEE 754 浮動小数点規格などに準じたものです。
https://ja.wikipedia.org/wiki/ゼロ除算
あたりを見れば様々な例外的な処理が考慮されていることがわかるでしょう。

Pythonなど、比較的積極的に NaN を扱うことが多い言語もあります。この機会に勉強してみると良いでしょう。

■ ARM CPU でのゼロ除算例外

多くのアーキテクチャ及び C 言語処理系においてゼロ除算例外は上述の振る舞いとなります。しかし ARM CPU は歴史的経緯からゼロ除算例外の処理が必ずしも働くとは限らない状況にあり、実際に ARM アーキテクチャの CPU を使った M1/M2 Mac では整数でのゼロ除算は驚いたことに！ゼロとなります。10/0 が 0 になるなんて！
これを何とかエラーとして検出させる方法はあるのですが、あまりこの導入クラスで扱うような内容ではありませんのでここではこれ以上説明しません。いつか実行環境の整理によってゼロ除算例外として検出されるようになることを期待します。興味のある人は「ARM CPU におけるゼロ除算例外」「divide by zero ARM M1 Mac」あたりで検索すると情報が得られると思います。