TP 5 : Data Representation

1 Float behaviour

1. First, open up the `float_equality.c` program and look through the code. This program compares `double`s using == - a seemingly-fine operation, but what happens when you compile it ? What output do you get ?
2. Run the program now changing the data type to `float` and examine the output. Was there a change in the output ? Why might this be ? What impact does that have on the arithmetic ? You may also want to see what the `float` values look like in the `float` visualizer here : click on https://www.h-schmidt.net/FloatConverter/IEEE754.html.
3. Try and see if you can find the min/max values for the different datatypes. How do the min/max values of the `float/double` range compare to the min/max of `int/long` ? Note the lack of symmetry in the definition names : INT_MAX/FLT_MAX are largest magnitude positive int/float, but INT_MIN is largest magnitude negative int whereas FLT_MIN is smallest magnitude positive float (normalized). What is the largest magnitude negative `float` ?
4. When an integer operation overflows, the result is wrapped back around the number circle. What is the result of a float operation that overflows ? An integer divide by zero halts the program. What about floating point divide by zero ? The float representation includes the exceptional float values infinity and NaN. What kind of calculations produce an exceptional result ? What happens when you use an exceptional value in further calculations ? Do the observed behaviors seem like reasonable choices to you ?

2 Software Bug : Acing every test

This is a true story of code encountered at a well-known online educational platform. Specifically, they had a function to check correct answers in their quizzes - we've provided an implementation resembling the original code below, and in `test.c`. However, their failure to account for exceptional float values created an embarrassing vulnerability in the implementation. The function worked by scoring quiz items with a numeric response using a "tolerance" feature that accepted all responses within a tolerance of the true answer as correct. For example, if the true answer was 200.7 and the tolerance was 0.01, responses within 1% of 200.7 would be scored as correct.

The code to score the student's quiz response looked something like this :

```
bool is_correct(const char *response, float target) {
/* Is student response within acceptable tolerance of target?

- response : student's answer (string)
- target   : instructor result (number)
*/
const float TOLERANCE = 0.01;
```

```
float val = strtof(response, NULL);
float at_most = TOLERANCE * fmax(fabs(val), fabs(target));
return fabs(val - target) <= at_most;
}
```

It turns out that there is a response that enabled users to get every answer right. What is the magic answer ? You can test your hypothesis by running the `test.c` program. It takes in one command-line argument and uses that as your "answer" to various hardcoded questions. Try different answers to see what you find !

3 Floating point addition

For this question, you are to complete the `float-skel.c` file (look at https://amritasuresh.github.io/teaching/).

Reminder : In C, the type `float` is represented according to the IEEE 754 standard in the 32 bit. 1 bit for the sign, 8 for the exponent and 23 for the mantisse.

```
typedef struct { int sign; int exponent; int mantissa; } fc;
```

1. Write a C function that decomposes a float into its three components. For example, the representation of 2.5 in IEEE 754 is :

   `0 . 1000 0000 . 010 0000 0000 0000 0000 0000`

   In this case, the returned structure would contain `sign` $= 0$, `exponent` $= 0x80 = 128$ and `mantissa` $=$ 0x200000 $= 2097152$.

   Reminder : To do this, it is convenient to use typecast.
2. Create a function that does the opposite, that is to say that returns the `float` corresponding to a given `fc` structure.
3. Realize the actual addition based on the addition of integers by going through the structures of `fc`. To simplify, we will make the following restrictions : (i) both operands are positive (ii) no special cases NaN / Inf etc.

   Addition in the `fc` type is done in three steps :

   (a) Standardize the two values, i.e. if the two exponents are different, we adjust the mantissa of one of the two according to the difference.

   (b) Add the sum of the two mantissae, taking into account the “hidden" bit representing the 1.

   (c) Normalize the mantissa for whatever is in $[1, 2)$, while adjusting the exponent of the result.

4 Mandelbrot revisited

We are again interested in the Mandelbrot fractal which associates with each pixel in an image a coordinate (x, y) to paint a colour that is a function of (x, y). Suppose that we are interested in zooming in on a specific point (x_0, y_0). For $\epsilon = 0.25$, we look therefore a series of images of 1024 x 1024 pixels where the image number i represents the interval $[x_0 - \epsilon^i, x_0 + \epsilon^i] \times [y_0 - \epsilon^i, y_0 + \epsilon^i]$. After a certain number of iterations, the image becomes blurred : the precision of the floating point representation is no longer sufficient to distinguish 1024 values different in width or height ; so some pixels will be associated with the same coordinates. Suppose first that the calculation is done with type `float`. To simplify, we are only interested in the width, so we set $x_0 = 0.375$ for any y_0.

Floating point formats consist of three components, the sign, the exponent and the mantissa. In what follows the sign will always be positive (0). For the `float` type, the exponent has 8 bits and the mantissa 23. So the binary representation of the exponent will be a an integer between 0 and 255, and we subtract 127 to get the exponent to use. (Special cases like 0, ∞, NaN will have no consequence in what follows.) In the mantissa, the most significant bit represents 0.5, and it will always be necessary to add 1. We consider the representation binary following a `float` (the dots simply indicate the limit between sign, exponent, mantissa) :

`0.011 1110 1.100 0000 0000 0000 0000 0000`

With the above explanations, this is the representation of $2^{125-127} \cdot (1 + 0.5) = 0.375$.

1. Give the binary representation of ϵ and ϵ^2. (You can omit the zeros that lag behind at the end.)
2. Give the binary representation of $x_0 + \epsilon^2$ and $x_0 + \epsilon^3$. (Be careful to choose the right exponent in each case.)
3. What is the representation of $x_0 - \epsilon^3$, knowing that $(x_0 + \epsilon^i) + (x_0 - \epsilon^i) = 2x_0$?
4. How many different values can we represent in the interval $[x_0 - \epsilon^3, x_0 + \epsilon^3]$? (A response of the form 2^n is enough.)
5. What will be the maximum value of i such that the precision is sufficient to represent 1024 = 2^10 different values in image number i ? And if the calculation was done with the `double` type, where the exponent has 11 bits and the mantissa 52 ?