System Programming in C – Homework Exercise 4

Publication date: *Thursday, May 21, 2020*
Due date: *Thursday, June 11, 2020 @ 21:00*

General overview:

- **Before you start**, copy over the source file `newFloat.c` from the shared directory `/share/ex_data/ex4/` to `~/exercises/ex4/`. You will be writing your code in this file, completing the implementation of seven functions.
- In each of the specific tasks in Problems 1 & 2 make sure to modify the code of `newFloat.c` only in the marked space.

Coding guidelines:

- Follow the implementation guidelines specified for each function. Not all guidelines are covered by the automatic tests, but they will be checked manually by the graders, so **make sure to stick to the guidelines**.
- Write clear and readable code. Use appropriate indentation and try to follow the style of `newFloat.c` as much as possible. You should also add brief documentation for the critical parts of each function's implementation. Keep in mind that your code is reviewed by the graders and 10 grade points will be allocated for code style.

Compilation instructions and testing:

- Make sure that your code compiles without errors or warnings and passes all the tests specified for each function.
- Testing your solutions requires you to compile your code with the test code file `/share/ex_data/ex4/test_ex4.c`. This file contains a main function, which runs tests for every function you implement. You "turn on" a specific test by compiling your source file together with the `test_ex4.c` and using the `-D TEST_<QUESTION_ID>` option, where `<QUESTION_ID>` is the question id. Example for testing (here `<QUESTION_ID>` is set to `1_3` for Problem 1.3):

```
==> gcc -Wall -D TEST_1_3 \
        /share/ex_data/ex4/test_ex4.c newFloat.c \
        -o test_ex4_1_3
```

[We discuss compilation of multi-file programs in detail in weeks #11 and #12]

- After compilation, you should run the program (`test_ex4_1_3` in the example above) and compare the output with the expected output provided in file `/share/ex_data/ ex4/test_ex4_<QUESTION_ID>.out`.

Problem 1:

The purpose of this question is to implement three functions for determining the bit contents of a given **`int`**. Recall that on our server, the compiler allocates 4 bytes (32 bits) for every **`int`**, but this size may be different in other systems. So you should not explicitly assume a specific size for **`int`** and use the **`sizeof()`** operator when appropriate. In your implementation of these functions, try to use simple expressions to compute the return value. In particular, do not use loops or function calls.

1. Complete the implementation of the function **`rightBits`**, such that it returns the right-most (least significant) bits of a given integer. The function receives two **`int`** parameters, **`n`** and **`numBits`**, and returns an **`int`** that is equal to the **`numBits`** least significant bits of **`n`** (). If **`numBits`** is non-positive, then the function should return 0, and if **`numBits`** is larger than the number of bits allocated for an **`int`**, then the function should return **`n`**.

   Execution examples: (on our server)

   - **`rightBits(7,2)`** returns 3
   - **`rightBits(7,5)`** returns 7
   - **`rightBits(7,80)`** returns 7
   - **`rightBits(7,-1)`** returns 0
   - **`rightBits(-1,2)`** returns 3 [think why this is]
   - **`rightBits(-1,7)`** returns 127 [think why this is]
   - **`rightBits(1,80)`** returns 1

   Validation: Test your code by compiling it with the test code file `/share/ex_data/ex4/test_ex4.c` using `<QUESTION_ID>=1_1` (see instructions on page 1). Execute the resulting program and compare your output with `/share/ex_data/ex4/test_ex4_1_1.out`.

   Manual inspection: your code should be correct for different sizes of **`int`** (e.g., if **`int`** takes 2 bytes), and it should not use loops or function calls.

2. Complete the implementation of the function **`leftBits`**, such that it returns the left-most bits of a given integer. The function receives two **`int`** parameters, **`n`** and **`numBits`**, and returns an **`int`** that is equal to the **`numBits`** left-most bits of **`n`**. Note that these bits are 0 if **`n`** does not occupy the **`numBits`** left-most bits in the space allocated for an **`int`**. If **`numBits`** is non-positive, then the function should return 0, and if **`numBits`** is larger than the number of bits allocated for an **`int`**, then the function should return **`n`**.

Execution examples: (on our server)

- **`leftBits(7,31)`** returns 3 [think why this is]
- **`leftBits(7,30)`** returns 1
- **`leftBits(7,29)`** returns 0
- **`leftBits(-1,2)`** returns 3
- **`leftBits(-1,7)`** returns 127
- **`leftBits(-1,80)`** returns -1
- **`leftBits(-1,-1)`** returns 0

Validation: Test your code by compiling it with the test code file `/share/ex_data/ex4/test_ex4.c` using `<QUESTION_ID>=1_2` (see instructions on page 1). Execute the resulting program and compare your output with `/share/ex_data/ex4/test_ex4_1_2.out`.

Manual inspection: your code should be correct for different sizes of **`int`** (e.g., if **`int`** takes 2 bytes), and it should not use loops or function calls.

3. Complete the implementation of the function **`getBit`**, such that it returns a specific bit in a given number. The function receives two **`int`** parameters, **`n`** and **`bitInd`**, and returns the value of the **`bitInd`**th bit (0 or 1) of **`n`**. For this purpose, the least significant bit (the parity bit) is associated with index 0, and the most significant bit (the sign bit) has the maximum index (determined by the size of **`int`**). If **`bitInd`** is out of bounds, then the function should return 0.

   Execution examples: (on our server)

   - **`getBit(7,0)`** returns 1
   - **`getBit(7,2)`** returns 1
   - **`getBit(7,3)`** returns 0
   - **`getBit(7,-12)`** returns 0
   - **`getBit(-404,31)`** returns 1 [think why this is]
   - **`getBit(-404,0)`** returns 0 [think why this is]
   - **`getBit(-404,32)`** returns 0

   Validation: Test your code by compiling it with the test code file `/share/ex_data/ex4/test_ex4.c` using `<QUESTION_ID>=1_3` (see instructions on page 1). Execute the resulting program and compare your output with `/share/ex_data/ex4/test_ex4_1_3.out`.

   Manual inspection: your code should be correct for different sizes of **`int`** (e.g., if **`int`** takes 2 bytes), and it should not use loops or function calls.

Final testing for Problem 1: Execute script `/share/ex_data/ex4/test_ex4.1` from the directory containing your `newFloat.c` source file.

(the script produces a detailed error report to help you debug your code)

Problem 2:

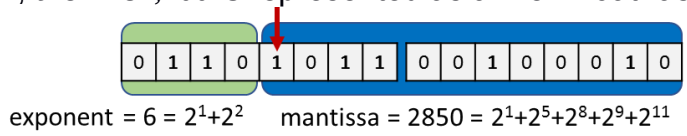
The purpose of this question is to implement a floating-point representation for large integers. This "new float" uses the same space as **`int`** (4 bytes on our server), but it enables representation of much larger integers, by sacrificing some accuracy. Our "new float" is similar in spirit to **`float`**, but it simpler in two main ways: (1) it does not represent negative numbers, and (2) it only represents integers. On the other hand, a nice feature of our "new float" is that we can control the number of bits allocated to the exponent. The representation is described in detail below. Read this carefully before you start implementing.

Specification of "new float":

- The e left-most bits hold the binary representation of the exponent, and the remaining m bits hold the binary representation of the mantissa. Both exponent and mantissa are considered as non-negative integers here.
- If the exponent (as determined by the e left bits) is equal to EXP, and the mantissa (as determined by the m right bits) is equal to MAN, then the value of the "new float" is considered to be $MAN \times 2^{EXP}$.
- Integer values that can be represented by m bits or less (meaning that they are in the range $[0, 2^m-1]$) are represented with complete accuracy using the mantissa bits and setting the exponent to 0.
- Integer values that are represented by $n>m$ bits (meaning that they are larger than 2^m-1), are represented by setting the mantissa to the m most significant bits of the number, and setting the exponent to $n-m$. This implies the following:
  - When the exponent is non-zero, the left-most bit of the mantissa is 1.
  - If an integer number is represented by $n>m$ bits and one of its $n-m$ least significant bits is non-zero, then its representation as a "new float" will be approximate and not accurate.

An example:

To demonstrate this specification, consider the specific example of the large integer $182{,}400$. We would like to represent this integer using a "new float" in a system where **`int`** takes up only two bytes (we assume this for simplicity). If we assume that $e=4$ and $m=12$, then $182{,}400$ is represented as a "new float" as follows:



Because the exponent is 6 and the mantissa is $2{,}850$, the value represented by this "new float" is $2{,}850 \times 2^6 = 2{,}850 \times 64 = 182{,}400$. Note that this integer is larger than $65{,}535=2^{16}-1$, so it cannot be represented by simple binary integer representation using two bytes. As specified above, the left-most bit of the mantissa is 1 (indicated by the red arrow). Note that the same integer value can also be represented as $182{,}400=1{,}425 \times 2^7$, but setting the mantissa to $1{,}425$ (shifting it one bit to the right)

will result in a 0 in the left-most bit of the mantissa, which violates the standard requirement mentioned above. Finally, note that this "new float" represents all values in the range $[182{,}400, 182{,}463]$, because the binary integer representations of numbers in this range takes up $n=18$ bits, and their $m=12$ most significant bits are identical to the mantissa specified in the figure above.

"New float" versus `float`: (… for those of you who are curious)

Our "new float" is actually very similar to the standard IEEE floating point representation used in C (and accepted widely by nearly all computer applications). Two obvious differences are that **`float`** also represents negative values (by allocating one bit for the sign), and fractional values (by allowing negative exponents). Another subtle difference has to do with the lead bit of the mantissa (the one indicated by an arrow in the figure in the previous page). In order to guarantee a standard representation, we require this bit to be set to 1 whenever the exponent is greater than 0. This is clearly a bit wasteful. In **`float`** this lead bit is implicitly assumed to be '1' without it actually being stored in memory, allowing the representation to gain an additional mantissa bit "for free."

Implementation notes:

The source file `newFloat.c`. contains five functions that deal with "new floats". In all functions, the "new float" value is represented by a simple **`int`**. We implemented one of these functions, **`newFloatToDouble`**, which takes an **`int`** representing a "new float" and returns its actual value as a **`double`**. You are encouraged to read this function for your reference. You should implement the remaining 4 functions.

The number of exponent and mantissa bits (e and m) are set using the **symbolic constants** **`NUM_EXP_BITS`** and **`NUM_MAN_BITS`** at the top of the file:

```
#define NUM_EXP_BITS 4
#define NUM_MAN_BITS (sizeof(int)*8 - NUM_EXP_BITS)
```

Use these two symbolic constants in your code when you wish to refer to e and m. Symbolic constants are not variables in your programs. They are "text replacement' directives for the compiler. The compiler will replace every instance of **`NUM_EXP_BITS`** in your code with 4, and every instance of **`NUM_MAN_BITS`** with **`(sizeof(int)*8 - NUM_EXP_BITS)`**, which evaluates on our server to 28 (= 4*8 – 4). We discuss symbolic constants in detail later in the course.

Note that if you want to change the number of exponent bits, you need to change the value of **`NUM_EXP_BITS`** from 4 to something else, and then you need to recompile the program. You are encouraged to try this and see how it affects the results of your tests in Problems 2.3 and 2.4.

1. Complete the implementation of the function **`getExponent`**, such that it returns the integer value of the exponent, given a new float. Do this by replacing the 0 in the return statement with the appropriate expression. You may invoke functions that you implemented in Problem 1 above.

   Validation: Test your code by compiling it with the test code file `/share/ex_data/ex4/test_ex4.c` using `<QUESTION_ID>=2_1` (see instructions on page 1). Execute the resulting program and compare your output with `/share/ex_data/ex4/test_ex4_2_1.out`.

   Manual inspection: your code should contain a single line with a return statement.

2. Complete the implementation of the function **`getMantissa`**, such that it returns the integer value of the mantissa, given a new float. Do this by replacing the 0 in the return statement with the appropriate expression. You may invoke functions that you implemented in Problem 1 above.

   Validation: Test your code by compiling it with the test code file `/share/ex_data/ex4/test_ex4.c` using `<QUESTION_ID>=2_2` (see instructions on page 1). Execute the resulting program and compare your output with `/share/ex_data/ex4/test_ex4_2_2.out`.

   Manual inspection: your code should contain a single line with a return statement.

3. Complete the implementation of the function **`getNewFloat`**, which creates a "new float" (as an **`int`**). The function receives two **`int`** parameters: **`mantissa`**, **`exponent`**, and returns the appropriate "new float". Follow these guidelines to ensure that your implementation adheres to the specification from page 4:

   - If **`mantissa`** takes up less than **`NUM_MAN_BITS`** bits and **`exponent`** is non-zero, then you need to adjust the values of the actual mantissa and exponent to represent the same value in the standard form (such that the left-most bit of the mantissa is 1 or the exponent is 0).
   - If **`mantissa`** takes up more than **`NUM_MAN_BITS`** bits, then the actual mantissa should consist of the **`NUM_MAN_BITS`** most significant bits of **`mantissa`**, and the actual exponent should be adjusted accordingly.
   - If the exponent (as specified in the parameter or after the adjustments mentioned above) takes up more than **`NUM_EXP_BITS`**, then the value is too large to be represented by "new float" and you should return the maximum "new float". **Hint:** the maximum "new float" has a simple representation as a signed **`int`**. Figure this out and simply return this **`int`**.

Execution examples: (on our server, with **`NUM_EXP_BITS`** set to 4)

- **`newFloatToDouble( getNewFloat(6,0) )`** returns 6.0
  (actual mantissa is 6 and actual exponent is 0)
- **`newFloatToDouble( getNewFloat(3,1) )`** returns 6.0
  (actual mantissa is 6 and actual exponent is 0)
- **`newFloatToDouble( getNewFloat(1000000,20) )`** returns 1.048576e12
  (actual mantissa is 256,000,000 and actual exponent is 12)
- **`newFloatToDouble( getNewFloat(1023456789,3) )`** returns 8.187654e+09
  (actual mantissa is 255,864,197 and actual exponent is 5)
- **`newFloatToDouble( getNewFloat(1000000,30) )`** returns 8.796093e+12
  (actual exponent is out of bounds, so function returns the maximum "new float", which under these circumstances is 8.796093e+12)

Validation: Test your code by compiling it with the test code file `/share/ex_data/ex4/test_ex4.c` using `<QUESTION_ID>=2_3` (see instructions on page 1). Execute the resulting program and compare your output with `/share/ex_data/ex4/test_ex4_2_3.out`.

4. Complete the implementation of the function **`newFloatSum`**, which computes the sum of two "new floats." The function receives two **`int`** parameters, **`newFloatNum1`** and **`newFloatNum2`**, and returns the "new float" that represents their sum. The sum can be computed as follows. Denote by MAN_1 and MAN_2 the two mantissas and by EXP_1 and EXP_2 the two exponents, and assume (without loss of generality) that $EXP_1 \geq EXP_2$ (otherwise you can switch their roles). Then, the sum of the two "new floats" can be represented as follows:

$$MAN_1 \times 2^{EXP_1} + MAN_2 \times 2^{EXP_2} \;=\; (MAN_1 + MAN_2/2^{EXP_1-EXP_2}) \times 2^{EXP_1}$$

This equation implies that by setting $EXP = EXP_1$ and $MAN = MAN_1 + MAN_2/2^{EXP_1-EXP_2}$, the sum can be represented as $MAN \times 2^{EXP}$. However, since MAN may be larger than $2^m - 1$, the actual exponent and mantissa may need to be adjusted. Further, note that if the adjusted exponent takes up more than **`NUM_EXP_BITS`**, then the sum is too large to be represented by "new float" and the function should return the maximum "new float" (see Problem 2.3 above). If one of the two parameters does not correspond to a valid "new float", the function should also return the maximum "new float." You may invoke functions that you implemented in Problems 1 and 2.1-2.3.

Execution examples: (on our server, with **`NUM_EXP_BITS`** set to 4)

- If **`a=b=6`** then **`newFloatSum(a,b)`** returns 12
- If **`a=6`** and **`b=100`** then **`newFloatSum(a,b)`** returns 106
- If **`a=-10`** and **`b=10`** then **`newFloatSum(a,b)`** returns -10
  (**`a`** represents a very large number (~8.8e+12) and **`b`** represents a much smaller number, so **`a+b`** ends up being rounded down to **`a`**.)
- If **`a=-2`** and **`b=1<<14`** then **`newFloatSum(a,b)`** returns -2
  (**`a`** represents a very large number (~8.8e+12) and **`b`** represents a much smaller number, so **`a+b`** ends up being rounded down to **`a`**.)

Cases where the maximum "new float" is returned:

- If **`a=-2`** and **`b=1<<15`** (**`a`** represents a very large number (~8.8e+12) and **`b`** is sufficiently large such that **`a+b`** equals the maximum "new float")
- If **`a=-2`** and **`b=1<<16`** (**`a+b`** is out of bounds)
- If **`a=(8<<28)|300`** and **`b=100`** (**`a`** is an invalid "new float" because it has a positive exponent (8) and the left-most bit of its mantissa (300) is 0)
- If **`a=getNewFloat(1023456789,3)`** and **`b=(5<<28)|1000`** (**`b`** is an invalid "new float" because it has a positive exponent (5) and the left-most bit of its mantissa (1000) is 0)

Validation: Test your code by compiling it with the test code file `/share/ex_data/ex4/test_ex4.c` using `<QUESTION_ID>=2_4` (see instructions on page 1). Execute the resulting program and compare your output with `/share/ex_data/ex4/test_ex4_2_4.out`.

Final testing for Problem 2: Execute script `/share/ex_data/ex4/test_ex4.2` from the directory containing your `newFloat.c` source file.

(the script produces a detailed error report to help you debug your code)

Submission Instructions:

1. After you validated and tested your solution, make sure that your `~/exercises/ex4/` directory contains the following C source file, which includes your implementation:
   - `newFloat.c`
2. your `~/exercises/ex4/` directory should also contain a `PARTNER` file with the user id of the non-submitting partner. The non-submitting partner should also add a `PARTNER` file containing the user id of the submitting partner.
3. Check your solution by running **`check_ex ex4`**. The script should be executed from the account of the submitting partner, and it may be run from any directory. Clean execution of this script guarantees you 80% of the assignment's grade.
4. Once you are satisfied with your solution, you may submit it by running **`submit_ex ex4`**. The script should be executed from the account of the submitting partner, and it may be run from any directory. You may modify your submission any time before the deadline (**11/6 @ 21:00**) by running **`submit_ex ex4 -o`** from any location.
5. For more information on the submission process, see the **Homework submission instructions** file on the course website.