Creating a Data Type in C — Rational Numbers

For this assignment, you will use the **`struct`** mechanism in C to implement a data type that represents rational numbers. A set can be modeled using the C **`struct`**:

```
struct _Rational {

   int64_t Top;        // numerator
   int64_t Bottom;     // denominator
};
typedef struct _Rational Rational;
```

A *rational number* is any number *Q* that can be represented as a ratio of two integers, *N* and *D*, such that *D* is not zero, so the **`struct`** type above seems to be entirely appropriate. We will say that a `Rational` object is *proper* if `Top` and `Bottom` have been initialized and `Bottom` is not 0.

How will we deal with representation issues? For example, all of the following rational numbers (written in the usual mathematical fashion) are equal:

$$\frac{-3}{4} = \frac{3}{-4} = \frac{-9}{12} = \frac{75}{-100}$$

Will we represent all of them the same way, or differently? If we represent them all the same way, what will that be? This can be characterized as a question of *normalization*; that is, will we adopt some standard (normal) scheme for representation?

How will we deal with a client who tries to create a `Rational` variable with denominator set equal to 0? Mathematically, such an expression violates the definition of a rational number. Implementing this in an OO language, we would probably consider responding by throwing an exception; that's not an option in C. So, what will we do? There are ways to force a runtime error…

These decisions are left to you; how I handle them in my implementation will be discussed in class, but you are not required to follow my lead. I will guarantee that I will not test your implementation by ever setting the denominator of an operand to 0.

A *data type* consists of a collection of values and a set of operations that may be performed on those values. For a rational number type, it would make sense to provide the common arithmetic operations; for example:

```
/**
 *   Compute the sum of Left and Right.
 *   Pre:
 *        Left and Right are proper.
 *   Returns:
 *        A proper Rational object X equal to Left + Right.
 */
Rational Rational_Add(const Rational Left, const Rational Right);
```

The design of `Rational_Add()` follows the expected semantics of addition; when you add two values, neither of those is modified and a new value is produced.

The design also avoids using pointers and (apparently) dynamic allocation. `Rational` objects are small, 16 bytes, and so it's fine to pass them into the function by copy and avoid the overhead of pointer accesses. The function must create a new `Rational` object and return it; since the function must always create exactly one object, there's no need or reason to allocate that object dynamically. Since dynamic allocation costs extra time at runtime, the design wisely avoids using dynamic allocation.

We will copy one aspect of an OO design; it's useful to provide a function that will create a new `Rational` object:

```
/**
 *  Creates and initializes a new Rational object.
 *  Pre:
 *        Denominator != 0
 *  Returns:
 *        A proper Rational object X that represents the rational number
 *        Numerator / Demonimator.
 */
Rational Rational_Create(int64_t Numerator, int64_t Denominator);
```

To some degree, this plays the role of a constructor in an OO implementation; the basic difference is that this is actually responsible for creating the new object (but not dynamically). In contrast, in a Java implementation, you would call `new` to allocate memory for the object dynamically and then call the constructor to initialize that memory correctly.

The other required functions are:

```
Rational Rational_Negate(const Rational R);
Rational Rational_AbsVal(const Rational R);
int64_t  Rational_Ceiling(const Rational R);

Rational Rational_Add(const Rational Left, const Rational Right);
Rational Rational_Multiply(const Rational Left, const Rational Right);
Rational Rational_Divide(const Rational Left, const Rational Right);

bool Rational_Equals(const Rational Left, const Rational Right);
bool Rational_LessThan(const Rational Left, const Rational Right);
```

A useful version of the `Rational` type would include many more functions, but we're trimming the requirements down to a minimal set that's sufficient to illustrate the essentials of implementing such a type.

Download the posted tar file for this assignment to your CentOS installation or rlogin, and unpack it. The tar file contains:

`driver.c`	C test driver (read the comments!)
`Rational.h`	header file for assigned function (do not modify!)
`Rational.c`	shell file for implementing your solution
`TestRational.h`	header file for testing/grading code (do not modify!)
`TestRational.o`	64-bit implementation of the testing/grading code
`README`	instructions for compiling and using the testing harness

The file `Rational.h` includes header comments for each of the required functions. Pay attention to the comments in the header file. All the stated pre- and post-conditions are part of the assignment; the test harness will always obey the stated preconditions*, and failure to satisfy post-conditions may result in deductions. Place your implementation in a file named `Rational.c`.

You should consider implementing additional "helper" functions. Those should be private to your C file, so make them **`static`**; they will be invisible to the testing code and never called directly from outside your file.

There is one implementation restriction: since there is no logical reason to use dynamic memory allocation, and no advantage in doing so, you are not allowed to call `malloc()` or `calloc()` or `realloc()` in your solution. You are, of course, allowed to use pointers.

* Well, maybe. In some situations, this will depend on whether your implementation of certain functions is correct. In particular, the test harness will use your `Rational_Set()` function internally, so any bugs in that will likely cause problems elsewhere, even if your `Rational_Set()` passes the explicit testing itself.

What to Submit

You will submit a single `.c` file, containing the implementations of the specified C functions and any private helper functions you write. Be sure to conform to the specified function interfaces.

The *Student Guide* and other pertinent information, such as the link to the proper submit page, can be found at:

http://www.cs.vt.edu/curator/

Pledge:

Each of your program submissions must be pledged to conform to the Honor Code requirements for this course. Specifically, you **must** include the following pledge statement in the submitted file:

```
//  On my honor:
//
//  - I have not discussed the C language code in my program with
//    anyone other than my instructor or the teaching assistants
//    assigned to this course.
//
//  - I have not used C language code obtained from another student,
//    the Internet, or any other unauthorized source, either modified
//    or unmodified.
//
//  - If any C language code or documentation used in my program
//    was obtained from an authorized source, such as a text book or
//    course notes, that has been clearly noted with a proper citation
//    in the comments of my program.
//
//  - I have not designed this program in such a way as to defeat or
//    interfere with the normal operation of the Curator System.
//
//    <Student Name>
//    <Student's VT email PID>
```

We reserve the option of assigning a score of zero to any submission that is undocumented or does not contain this statement.

Change Log

Version	Posted	Pg	Change
4.00	Mar 29		Base document.

Some Design and Implementation Considerations

Designing a data type requires considering the needs and likely behavior of the programmer who will use the type. You should ask questions like:

- What range of values would users want to use?
- What operations would users want to apply to those values?
- What common errors should I expect, and how should I deal with them?

The desired range of values for a numeric type like `Rational` is probably "as large as possible", which would argue for using the "widest" integer types available. That must be balanced with a consideration of the memory cost of using wider types. In this case, I decided to use 64-bit integers because that will maximize the effective range of the type, as long as we restrict ourselves to standard C types. I'm convinced you could simply replace **`int64_t`** with **`int32_t`** in a working implementation and it would compile and still work.

Another range consideration is whether the values involved are signed or unsigned (i.e., can they be both negative and nonnegative, or are they always one or the other)? Obviously, rational numbers can be negative or nonnegative; hence, we use a signed integer type for the numerator and denominator. But… see the normalization discussion below.

As for operations, any common arithmetic operation would seem to be in play. I would include addition, subtraction, multiplication, division (watch for divide-by-zero issues), floor, ceiling, perhaps another rounding operation, negating, perhaps absolute value, taking reciprocals, and probably others. A good type implementation will include every operation that makes sense, even if it's redundant (like reciprocals); that makes the type as convenient as possible for the user.

I would also include functions to return the values of the numerator and denominator, even though (in C) the user could access them directly. In general, I try to use **`struct`** types in C in much the same way as if I had a class in Java or C++.

As for user errors… the obvious one is to specify a rational number whose denominator is zero. That makes no sense mathematically. In C++ or Java I'd consider responding with an exception. In C, I could generate a forced runtime error (e.g., write `*NULL = 0`), or simply override the user's input with a correction (obnoxious since it's invisible), or I could decide to allow a representation like 1/0 since it doesn't actually entail a division on my part.

Another error would be division by 0. 0 is a perfectly good rational number (as are all integers); but you cannot divide by it. Nevertheless, a user might attempt to do so accidentally. I'd deal with this as described above.

Normalization

Normalization refers to the adoption of some standardization for how data will be represented. In this case, I considered two such issues:

- How should I deal with negative values?
- Should I represent rational values in a reduced form? That is, should I only store 2/3 or should I allow representations like 16/24?

For the first question, I decided that I would never store a negative value for the denominator. So, I would store -5/7 instead of 5/-7, and I would store 9/32 and never -9/-32. That complicates my initializer function a bit, but it simplifies some arithmetic issues in some other functions; it also makes it easier to display a `Rational` variable in a uniform way.

For the second question, I decided that I would always reduce rationals to their simplest form; so I store 36/42 as 6/7. This led me to implement a private (**`static`**) helper function that would compute the *greatest common divisor* (GCD) of two integers. There are several algorithms for doing this; two implementations of Euclid's Algorithm are shown on my C notes on Recursion:

```
uint64_t GCD(uint64_t x, uint64_t y) {

   if ( y == 0 ) return x;

   return GCD(y, x % y);
}
```

```
uint64_t GCD(uint64_t x, uint64_t y) {

   while ( y != 0 ) {
      uint64_t Remainder = x % y;
      x = y;
      y = Remainder;
   }
   return x;
}
```

Feel free to incorporate either of these in your solution; I recommend the iterative version. Do note that the parameters are specified as unsigned integers, so you need to be sure you call the function with the correct values. There is an `abs()` function in `stdlib.h`, however it may yield incorrect results when passed a value outside the range of the `int32_t` type. Use the function `llabs()` instead.

In this case, I had two pressing reasons to reduce the rational numbers to lowest terms. First, I wanted the representation to be easy for human interpretation; it's much easier to understand 17/32 than 73967/139232. Second, I decrease the risk of getting an integer overflow when doing operations like addition or multiplication. If overflow happens, I will get incorrect results.

When considering the implementation, I realized that if I wanted to keep my `Rational` objects reduced to lowest terms, I'd have to apply that logic at more than one place in my code. That implied using another private helper function that I could just call as needed to handle the process.

Whether, and how, you normalize your `Rational` objects is up to you. The test code I use doesn't care about that. But, you may encounter overflow issues if you don't.