Homework 2 - MPCS 51040

(last modified: October 30, 2014)

Issued: October 27, 2014

1 General Instructions

1.1 Compiling

Your code must compile with `gcc -std=c11 -Wall -pedantic`. There should be no warnings or errors when compiling with gcc-4.8 (as installed on `linux.cs.uchicago.edu`).

1.2 Handing in

To hand in your homework, you need to commit all requested files (with correct filenames!) to your personal subversion repository. You should have access to a repository named yourcnetid-mpcs51040-aut-14.

Make a subdirectory called 'homework/hw2' and place your files under that directory. Don't forget to commit your files! You can check on `http://phoenixforge.cs.uchicago.edu` to make sure all files were committed to the repository correctly.

If you can cannot login to `http://phoenixforge.cs.uchicago.edu` or have issues accessing your repository, please contact `techstaff@cs.uchicago.edu`. Please cc `dries@uchicago.edu`.

The deadline for this homework is 5:30pm, November 10, 2014. To grade the homework, the contents of your repository at exactly the deadline will be considered. Changes made after the deadline will be ignored.

1.3 Code samples

This document, and any file you might need to complete the homework can be found the subversion repository `https://wooldridge.cs.uchicago.edu/svn/mpcs51040-aut-14`.

1.4 Grading

Your code will be graded based on the following points (in order of descending importance):

- Correctness of the C code: there should be no compiler errors or warnings when compiling as described in 1.1. There should be no memory leaks or other problems (such as those detected by valgrind).
- Correctness of the solution. Your code should implement the required functionality, as specified in this document.
- Code documentation. Properly documented code will help understand and grade your work.
- Code quality: your code should be easy to read and follow accepted good practices (avoid code duplication, use functions to structure your program, ...). This includes writing portable code (which will work on both 32 bit and 64 bit systems).
- Efficiency: your code should not use more resources (time or space) than needed.

Some items might be marked as 'optional' or 'extra credit'. When correctly completing these tasks, the points obtained could go towards mistakes made elsewhere in the same homework, possibly raising your grade.

2 Assignment

2.1 Problem Description

For this homework, we will design and implement an arbitrary precision integer math library. The integer types provided by the C language are fixed precision (for a given platform). For example, while `long int` might be 32 bit or 64 bits depending on the platform, the range of the type is limited and determined at compile time. The arbitrary precision integers to be provided by your library should be able to handle any integer, as long as enough memory is available to store them.

The homework will have two parts:

- Design the header file and implementation.
- Write a program exercising the functionality provided by your library and validating correct functionality.

2.2 Task 1 - Library Design and Implementation

Your library should satisfy the following design constraints:

- The basic type representing an arbitrary precision integer should be called `arb_int_t`. This does not have to be a value type, i.e. an variable of `arb_int_t`itself does not have to contain the arbitrary precision integer. (That would be difficult, because you might recall that in C, a datatype and thus variable has a fixed size, known at compile time. It would be difficult to store an arbitrary number of digits in a fixed size variable). This means that arbitrary precision integer variables cannot be copied or created by simply declaring them, or assigning them using '=', similar to how a C string (`char *`) cannot needs to be copied using `strcpy` instead of simple assignment, as the data type itself (`char *`) only holds a pointer to the actual memory containing the string. Copying the variable using '=' only copies the pointer.

  In a way, you could consider `arb_int_t`to be a handle representing the arbitrary precision integer. While you can copy the handle using simple assignment, that does not copy the arbitrary precision integer it refers to and merely causes both variables (handles) to refer to the same underlying unbounded integer.

  This is a common idiom in C. For example, `FILE *` exhibits the same behaviour.

  Because of this, you need explicitly need to provide functions to copy and destroy (free) the memory associated with `arb_int_t`variables.

```
// Free arb_int_t
void arb_free (arb_int_t i);

// Duplicate arb_int_t. The returned arb_int_t needs to be freed
// using arb_free.
arb_int_t arb_duplicate (const arb_int_t i);
```

- Since we can't directly create an arbitrary precision integer by simply declaring a variable of `arb_int_t`, we will need to provide a way to create a valid instance of our integer. One way to do this is to supply conversion functions that create and return an arg integer given a regular integer or string.

```
// Convert from string representation into arb_int_t.
// Return non-zero and set *i to the new arb_int_t if successful,
// do not change *i and return zero otherwise.
int arb_from_string (arb_int_t * i, const char * s);

// Convert from long int to arb_int_t
//   Return non-zero on success, zero otherwise.
int arb_from_int (arb_int_t * i, signed long int source);

// Convert arb_int_t to string
//   Return non-zero on success, or zero if buf is not big enough.
```

```
//   Make sure buf is always properly zero-terminated.
int arb_to_string (const arb_int_t i, char * buf, size_t max);

// Convert arb_int_t to long int
//   Return non-zero on success, zero otherwise.
int arb_to_int (const arb_int_t i, long int * out);
```

- Now that we can create and destroy arb integers, we need to provide functions to perform useful calculations on them.

```
// Add one arb_int_t to another one, modifying one of them.
// (x += y)
void arb_add (arb_int_t x, const arb_int_t y);

// Subtract an arb_int_t from another.
// (x -= y);
void arb_subtract (arb_int_t x, const arb_int_t y);

// Multiply
// x *= y
void arb_multiply (arb_int_t x, const arb_int_t y);

// Divide  (for extra credit)
// x /= y
// Note: this is integer division, so the result gets truncated to an
// integer. For example, 192/100 => 1
```

- Just like regular integers, `arb_int_t`numbers can be ordered.

```
// Return -1 if x<y, 0 if x==y and 1 if x>y.
int arb_compare (const arb_int_t x, const arb_int_t y);
```

Implement the public part of your library (i.e. the header) in a file named `arb_int.h`. The implementation of of these functions should go in a file name `arb_int.c`.

Note: while it is up to you to decide how to implement the required functionality, it is vital that you follow the function and file names as described in this section exactly (including case!). This will make it possible to use your test program (described in task 2) with any library implementation and vice-versa.

2.3 Task 2

You will likely work on task 2 and task 3 concurrently, as task 2 will enable you to test and validate the library you are implementing. The goal of task 2 is to create a program which will call your library functions. Call your program `arb_test.c`.

It has to have two modes:

- When called without command line arguments, it should run your internal testing making sure to call every function provided by `arb_int.h`, exercising and validating as much functionality as possible.
- When called with command line arguments, your program should react as follows:
  - 3 command line arguments: `number operation number`
    where number is an arbitrary precision integer and operation should be one of '+', '-', '*' or (optional) '/'. The program should output the resulting arbitrary precision integer on standard output, and only the result (do not output anything else).

    If an error occurred (invalid operation or invalid number was specified) your program should output an appropriate error message to standard error.

- More or less arguments: finish the program returning a non-zero exit code after having written the instructions on how to properly execute the program to standard error (`stderr`).

As explained during class, you do not need to create a real library to which your program will link. Instead we will compile both the library file and your test program at the same time, letting the compiler take care of the linking for us (as we have been doing so far).

```
prompt% ls
arb_test.c  arb_int.h   arb_int.c
prompt% gcc -std=c11 -Wall -pedantic -o arb_test arb_int.c arb_test.c
prompt% ./arb_test 3 + 99
102
prompt% ./arb_test 213437462837423433423423432423434 * 343423423412312231231232
732994241720661245375159437620722356168175410320759753748
prompt% ./arb_test 23s23 * 2
Error: '23s23' is not a valid integer!
```

2.4 Task 3

Create a file called 'README.TXT' which briefly describes how you implemented the requested functionality. Describe how you decided to encode an `arb_int_t`in memory, and how you implemented addition, subtraction etc.

Please be clear and complete; In case of accidental mistakes or bugs in your program, this file can help the person grading your work understand your intentions.

Do not underestimate the importance of the README file!