

# Homework 5 - MPC5 51040

(last modified: April 24, 2017)

Issued: April 24, 2017

## 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` (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 git repository.

Make a subdirectory called ‘homework/hw5’ and place your files under that directory. Don’t forget to commit and push your files! You can check on <http://mit.cs.uchicago.edu> to make sure all files were committed to the repository correctly.

The deadline for **the first part of this homework is Monday 5:30pm, May 1, 2017**. The deadline for **the complete homework is Monday 5:30pm, May 8, 2017**. To grade the homework, the contents of your repository at exactly the deadline will be considered. Changes made after the deadline are not taken into account.

### 1.3 Code samples

This document, and any file you might need to complete the homework can be found in the git repository <https://mit.cs.uchicago.edu/mpcs51040-spr-17/mpcs51040-spr-17/raw/master/tree/master/homework/hw5/hw5.pdf>.

### 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 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 new arb_int_t, when no longer needed,
// needs to be freed using arb_free. The original arb_int_t
// is unmodified.
// Returns zero on success, non-zero otherwise
int arb_duplicate (arb_int_t * new, const arb_int_t original);
```

- 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 `arb_int_t` given a regular integer or string.

```
// Convert from string representation into arb_int_t.
// Return zero and set *i to the new arb_int_t if successful,
// do not change *i and return non-zero otherwise.
// The following forms should be accepted:
// +2323 -232323 +0 -0 23232
// If any other characters than 0-9,+,- are present you should
// return an error (i.e. nonzero).
int arb_from_string (arb_int_t * i, const char * s);
```

```

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

// Convert arb_int_t to string
// Return zero on success, or non-zero if buf is not big enough.
// Make sure buf is always properly zero-terminated.
// max is the maximum number of bytes (including the terminating
// zero) than can be written to buf.
//
// Negative numbers should have a leading '-'; Positive numbers
// *should not* have a +. 0 is considered to be positive.
int arb_to_string (const arb_int_t i, char * buf, size_t max);

// Convert arb_int_t to long long int
// Return zero on success, non-zero otherwise.
// (for example if i > LLONG_MAX or < LLONG_MIN)
int arb_to_int (const arb_int_t i, long long int * out);

```

- The same thing is true for assigning one `arb_int_t` to another `arb_int_t`; Just like in the case of `FILE *`, simply doing `a=b` would not have the desired effect.

```

// Assign the numeric value of arb_int_t y to arb_int_t x. x and y have to be
// separate arb_int_t's constructed by one of the functions above.
// Note that
// x and y remain separate (i.e. changing x does not change y).
// This function differs from arb_duplicate in that both x and y
// have to be created already *before* calling this function.
// Returns 0 if success, return non-zero otherwise. In the latter case
// x and y remain unmodified.
int arb_assign (arb_int_t x, const arb_int_t y);

```

- 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)
// Return zero on success, non-zero otherwise
int arb_add (arb_int_t x, const arb_int_t y);

// Subtract an arb_int_t from another.
// (x -= y);
// Return zero on success, non-zero otherwise
int arb_subtract (arb_int_t x, const arb_int_t y);

// Multiply
// x *= y
// Return zero on success, non-zero otherwise
int 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
// Return 0 on success, non-zero otherwise

```

```
int arb_divide (arb_int_t x, const arb_int_t y);
```

- 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 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` at least once, exercising and *validating* as much functionality as possible. For example, you could create a number of `arb_int_t` instances (from integer, from string, duplicating them) and then use those instances to test addition, multiplication, subtraction and optionally division, followed by converting back to string and/or integer while verifying that the outcome is as expected.
- 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 (as in a `.a` archive) 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 * 3434234234123122312312322
732994241720661245375159437620722356168175410320759753748
prompt% ./arb_test 23s23 \* 2
Error: '23s23' is not a valid integer!
```

Note that you have to protect `*` from shell expansion (either by prepending it with `\` or by surrounding it with single quotes).

## 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!**

## 3 Timeline

This will be a two-week homework; However, part of the work has to be completed by week 1.

The following needs to be committed in the first week and will be graded at that point:

- Makefile, `arb_test.c`, `arb_int.h`, `arb_int.c`
- The following functions must be implemented (and testing in your `arb_test.c`):
  - `arb_free`
  - `arb_duplicate`
  - `arb_from_string`
  - `arb_from_int`
  - `arb_to_string`
  - `arb_to_int`
  - `arb_assign`

The remaining functions must be committed by the final deadline. Note that it is permitted to change your implementation (including the code already submitted) between week 1 and week 2.

## 4 Hints

- Your first task should be to read and understand the requested function prototypes. What is their goal? What are they supposed to do?
- Next, think of how you will represent your `arb_int_t` type. Once you have a representation in your mind, write down the C type corresponding with it.
- Does the C type you designed *fit* the required function prototypes? I.e. will it allow you to implement each requested function? **Remember, you are not allowed to change the listed prototypes in any way, so you will have to adjust your internal representation/C type to match the function prototypes!**
  - In particular, pay close attention to the access those functions will have to your type.
- **You are not allowed to use global variables in `arb_int.c` and `arb_int.h`.**