Computer Systems (SS 2013)
Exercise 2: April 22, 2013

Wolfgang Schreiner
Research Institute for Symbolic Computation (RISC)
Wolfgang.Schreiner@risc.jku.at

March 7, 2013

The exercise is to be submitted by the denoted deadline via the submission interface of the Moodle course as a single file in zip (`.zip`) or tarred gzip (`.tgz`) format which contains the following files:

- A PDF file `Exercise`*`Number`*`-`*`MatNr`*`.pdf` (where *Number* is the number of the exercise and *MatNr* is your "Matrikelnummer") which consists of the following parts:
  1. A decent cover page with the title of the course, the number of the exercise, and the author of the solution (identified by name, Matrikelnummer and email address).
  2. For every source file, a listing in a *fixed width font*, e.g. `Courier`, (such that indentations are appropriately preserved) and an appropriate *font size* such that source code lines do not break.
  3. A description of all tests performed (copies of program inputs and program outputs) explicitly highlighting, if some test produces an unexpected result.
  4. Any additional explanation you would like to give. In particular, if your solution has unwanted problems or bugs, please document these explicitly (you will get more credit for such solutions).
- Each source file of your solution (no object files or executables).

Please obey the coding style recommendations posted on the course site.

Exercise 2: Arbitrary-Precision Arithmetic

Write a program that implements arbitrary-precision arithmetic on natural numbers. The core of this program is a class `Nat` whose objects represent natural numbers of arbitrary size. It shall be possible to execute the following commands:

```
// initialization to 0, by another number, by a string of decimal digits
Nat a;
Nat b = a;
Nat c = "12345678901234567890";

// assignment by another number and by a string of decimal digits
a = c;
b = "0001234567890";

// representation as a null-terminated string of decimal digits
std::cout << b << std::endl; // "1234567890"

// addition and multiplication (by school algorithm)
Nat d = a+b;
Nat e = a*b;
std::cout << d << " " << e << std::endl;

// comparison operations
if (a == b) std::cout << "equal";
if (a <= b) std::cout << "less than equal";
```

The internal representation of a `Nat` object contains (among other information) a pointer to a heap-allocated byte array

```
typedef unsigned char byte;
byte* digits;
```

where every element represents *two decimal digits* of the number (i.e. the natural number is stored in the base 100 system) with the digits of lowest weight stored in `digits[0]`.

This array is not longer than is actually required, i.e., unless the number is 0 (in this case there is exactly one digit 0), the highest digit is not 0. This means that the result of a computation is first constructed in a temporary array whose size is an upper bound of the required size; once the actual size of the result is known, the result is copied into a new array of this size (and the temporary array is freed).

Please note the following:

- The byte array held by a `Nat` object is not shared by any other `Nat` object; if a `Nat` object is duplicated, a new byte array must be created for the new object.
- No memory leaks shall arise from the implementation. As a consequence,
  - the class needs a destructor that frees the memory allocated for the number when the object is destroyed,
  - the memory allocated for a number must be freed before the number gets assigned a new value.
- The sum of two natural numbers of maximum length n has at most length $n + 1$; the product of these numbers has at most length $2n$.
- All functions operating on `Nat` objects shall receive *references* to these objects as arguments such that no temporary duplicates are created. For instance, the operator + has signature

  ```
  Nat operator+(const Nat& a, const Nat& b);
  ```

- The definition of the operator << has structure

  ```
  std::ostream& operator<<(std::ostream& s, Nat const& v) {
    ...
    return s;
  }
  ```

  This operator is declared *outside* of `Nat`; if `Nat` contains a declaration

  ```
  friend std::ostream& operator<<(std::ostream& s, Nat const& v);
  ```

  then the operator has access to the internal representation of `Nat`.

Avoid any code duplication but make extensive use of auxiliary functions (that shall become as far as possible private member functions of `Nat`).

Write the declaration of `Nat` and of `operator<<` into a file `Nat.h` and the implementation of all member functions of `Nat` and of `operator<<` into a file `Nat.cpp`.

Write a file `NatMain.cpp` that uses `Nat` and tests its operations. Test each operation with at least three test cases that also include special cases (such as addition by 0).

In this exercise floating point arithmetic is neither needed nor allowed; fundamental arithmetic operations needed for this exercise are division by 100 and the remainder of this division.