$Id: asg2-dc-bigint.mm,v 1.67 2014-04-10 14:05:15-07 - - $
PWD: /afs/cats.ucsc.edu/courses/cmps109-wm/Assignments/asg2-dc-bigint
URL: http://www2.ucsc.edu/courses/cmps109-wm/:/Assignments/asg2-dc-bigint/

1. Overview

This assignment will involve overloading basic integer operators to perform arbitrary precision integer arithmetic in the style of `dc`(1). You will also perform explicit memory management using `new` and `delete`, and eliminate memory leak. Your class `bigint` will intermix arbitrarily with simple integer arithmetic.

To begin read the `man`(1) page for the command `dc`(1):

```
man -s 1 dc
```

A copy of that page is also in this directory. Your program will use the standard `dc` as a reference implemention and must produce exactly the same output for the commands you have to implement:

```
+ - * / % ^ c d f p q
```

Also look in the subdirectory `misc/` for some examples and in `output/` for the result of running the test data using `dc`.

2. Implementation strategy

As before, you have been given starter code.

(a) `Makefile`, `trace`, and `util` are similar to the previous program. If you find you need a function which does not properly belong to a given module, you may add it to `util`.

(b) The module `scanner` reads in tokens, namely a `NUMBER`, an `OPERATOR`, or `SCANEOF`. Each token returns a `token_t`, which indicates what kind of token it is (the `terminal_symbol symbol`), and the `string lexinfo` associated with the token. Only in the case of a number is there more than one character. Note that on input, an underscore (_) indicates a negative number. The minus sign (-) is reserved only as a binary operator. The scanner also has defined a couple of `operator<<` for printing out scanner results in `TRACE` mode.

(c) The main program `main.cpp`, has been implemented for you. For the six binary arithmetic functions, the right operand is popped from the stack, then the left operand, then the result is pushed onto the stack.

(d) The module `iterstack` can not just be the STL `stack`, since we want to iterate from top to bottom, and the STL `stack` does not have an iterator. A stack depends on the operations `back()`, `push_back()`, and `pop_back()` in the underlying container. We could use a `vector`, a `deque`, or just a `list`, as long as the requisite operations are available.

3. Class `bigint`

Then we come to the most complex part of the assignment, namely the class `bigint`. Operators in this class are heavily overloaded.

(a) Note that most of the functions take a right argument of type `const bigint &`, that is a constant reference, for the sake of efficiency. But they have to return the result by value.

(b) We want all of the operators to be able to take either a `bigint` or a `long` as either the left or right operand.

(c) When the left operand is an `long`, we make them non-member operators, with two arguments, the left being an `long` and the right a `bigint`. Note that the implementation of these functions, `+`, `-`, `*`, `/`, `%`, `==`, `<`, are all identical, namely that they cast the `long` to a `bigint` and call the other operator. This is unnecessary for `!=`, `<=`, `>`, `>=`, since they are inline functions, and given a `long` on either side, the non-explicit constructor `bigint (const long)` will cause an implicit conversion.

(d) The `operator<<` can't be a member since its left operand is an `ostream`, so we make it a `friend`, so that it can see the innards of a `bigint`. Note now `dc` prints really big numbers.

(e) The `pow` function exponentiates in $O(\log_2 n)$ and need not be changed. It is not a member of `bigint`, but it behaves as a member, since it uses only other functions.

(f) The relational operators `==` and `<` are coded individually as member functions. The others, `!=`, `<=`, `>`, and `>=` are defined in terms of the essential two.

(g) The `/` and `%` functions call `divide`, which is private. One can not produce a quotient without a remainder, and vice versa, so it returns a pair which is both the quotient and remainder, and the operator just discards the one that is not needed.

(h) The given implementation works for small integers, but overflows for large integers.

4. Representation of a bigint

Now we turn to the representation of a `bigint`, which will be represented by a boolean flag and a vector of integers.

(a) Replace the declaration

```
long long_value;
```

with

```
typedef unsigned char digit_t;
typedef vector<digit_t> bigvalue_t;
bool negative;
bigvalue_t big_value;
```

in `bigint.h`.

(b) In storing the long integer it is recommended that each digit in the range 0 to 9 is kept in an element, although true `dc`(1) stores two digits per byte. But we are not concerned here with extreme efficiency. Since the arithmetic operators add and subtract work from least significant digit to most significant digit, store the elements of the vector in the same order. That means, for example, that the number 4629 would be stored in a vector v as: $v_3 = 4$, $v_2 = 6$, $v_1 = 2$, $v_0 = 9$. In other words, if a digit's value is $d \times 10^k$, then $v_k = d$.

(c) In order for the comparisons to work correctly, always store numbers in a canonical form: After computing a value from any one of the six arithmetic operators, always trim the vector by removing all high-order zeros. While `size()` is > 0 and `back()` returns zero, `pop_back()` the high order digit. Zero should be represented as a vector of zero length and a positive sign.

(d) The representation of a number will be as follows: `negative` is a flag which indicates the sign of the number; `big_value` contains the digits of the number.

(e) Then use `grep` or your editor's search function to find all of the occurrences of `long_value`. Each of these occurrences needs to be replaced. Change all of the constructors so that instead of initializing `long_value`, they initialize the replacement value.

(f) The scanner will produce numbers as `string`s, so scan each string from the end of the string, using a `const_reverse_iterator` (or other means) from the end of the string (least significant digit) to the beginning of the string (most significant digit) using `push_back` to append them to the vector.

5. Implementation of Operators

(a) Add two new private functions `do_bigadd` and `do_bigsub`.

(b) Change `operator+` so that it compares the two numbers it gets. If the signs are the same, it calls `do_bigadd` to add the vectors and keeps the sign as the result. If the signs are different, call `abs_compare` to determine which one is larger, and then call `do_bigsub` to subtract the larger minus the smaller. Note that this is a different comparison function which compares absolute values only. Avoid duplicate code wherever possible.

(c) The `operator-` should perform similarly. If the signs are different, it uses `do_bigadd`, but if the same, it uses `do_bigsub`.

(d) To implement `do_bigadd`, create a new `bigvalue_t` and proceed from the low order end to the high order end, adding digits pairwise. If any sum is >= 10, take the remainder and add the carry to the next digit. Use `push_back` to append the new digits to the `bigvalue_t`. When you run out of digits in the shorter number, continue, matching the longer vector with zeros, until it is done. Make sure the sign of 0 is positive.

(e) To implement `do_bigsub`, also create a new empty vector, starting from the low order end and continuing until the high end. In this case, if the left number is smaller than the right number, the subtraction will be less than zero. In that case, add 10, and set the borrow to the next number to −1. You are, of course, guaranteed here, that the left number is at least as large as the right number. After the algorithm is done, `pop_back` all high order zeros from the vector before returning it. Make sure the sign of 0 is positive.

(f) You will need to implement `do_bigless`, which will compare the absolute values of the vectors to determine which is larger.

(g) To implement `operator==`, check to see if the signs are the same and the lengths of the vectors are the same. If not, return false. Otherwise run down both vectors and return false as soon a difference is found. Otherwise return true.

(h) To implement `operator<`, remember that a negative number is less than a positive number. If the signs are the same, for positive numbers, the shorter one is less, and for negative nubmers, the longer one is less. If the signs and lengths are the same, run down the parallel vectors from the high order end to the low order end. When a difference is found, return true or false, as appropriate. If no difference is found, return false.

(i) Implement function `do_bigmul`, which is called from `operator*`. `Operator*` uses the rule of signs to determine the sign of the result, and calls `do_bigmul` to compute the product vector.

(j) Multiplication in `do_bigmul` proceeds by allocating a new vector whose size is equal to the sum of the sizes of the other two operands. If $\mathbf{u}$ is a vector of size m and $\mathbf{v}$ is a vector of size n, then in $O(mn)$ speed, perform an outer loop over one argument and an inner loop over the other argument, adding the new partial products to the product $\mathbf{p}$ as you would by hand. The algorithm can be described mathematically as follows:

$$
\begin{array}{l}
\mathbf{p} \leftarrow \Phi \\
\textbf{for}\ \ i \in [0, m): \\
\quad c \leftarrow 0 \\
\quad \textbf{for}\ \ j \in [0, n): \\
\quad\quad d \leftarrow \mathbf{p}_{i+j} + \mathbf{u}_i \mathbf{v}_j + c \\
\quad\quad \mathbf{p}_{i+j} \leftarrow d \,\%\, 10 \\
\quad\quad c \leftarrow \lfloor d \div 10 \rfloor \\
\quad \mathbf{p}_{i+n} \leftarrow c
\end{array}
$$

Note that the interval $[a, b)$ refers to the set $\{x \mid a \le x < b\}$, i.e., to a half-open interval including a but excluding b. In the same way, a pair of iterators in C++ bound an interval.

(k) Long division is complicated if done correctly. See a paper by P. Brinch Hansen, "Multiple-length division revisited: A tour of the minefield", *Software — Practice and Experience 24*, (June 1994), 579–601. Algorithms 1 to 12 are on pages 13–23, Note that in Pascal, array bounds are part of the type, which is not true for `vectors` in C++.
`multiple-length-division.pdf`
`http://brinch-hansen.net/papers/1994b.pdf`
`http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.14.5815`

(l) The function `divide` as implemented uses the ancient Egyptian division algorithm, which is slower than Hansen's Pascal program, but is easier to understand. Replace the `long` values in it by `vector<digit_t>`. The logic is shown also in [`misc/divisioncpp.cpp`]. The algorithm is rather slow, but the big-O analysis is reasonable.

(m) Modify `operator<<`, first just to print out the number all in one line. You will need this to debug your program. When you are finished, make it print numbers in the same way as `dc`(1) does.

(n) The `pow` function is not a member and uses other operations to raise a number to a power. If the exponent does not fit into a single `long` print an error message, otherwise do the computation.

6. Memory leak

Make sure that you test your program completely so that it does not crash on a Segmentation Fault or any other unexpected error. Then implement the destructor `~bigint` so that there is no memory leak. But if you don't have time to do this, remember that memory leak is not as bad as a core dump. Use `valgrind` to check for and eliminate uninitialized variables and memory leak.

7. What to submit

Submit source files and only source files: `Makefile`, `README`, and all of the header and implementation files necessary to build the target executable. If `gmake` does not build `ydc` your program can not be tested and you lose 1/2 of the points for the assignment. Use `checksource` on your code.

If you are doing pair programming, follow the additional instructions in `Syllabus/pair-programming/` and also submit `PARTNER`.