$Id: asg2-dc-bigint.mm,v 1.6 2011-01-25 13:00:43-08 - - $
/afs/cats.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 **int** 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
```

You may also choose to print something with the debug function **Y**, but this is optional.

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.cc**, 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** makes use of the STL **deque** (double ended queue). We want to iterate from top to bottom, and the STL **stack** does not have an iterator. In order for iteration to work in the correct order, we push and pop from the front of the deque. Note that **deque.pop_front** does not return a value. This class uses the term “front” instead of the more usual “top” when referring to the elements of a stack. A **deque** is more efficient than a **vector** since we do not need any direct access to the elements.

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 an **int** as either the left or right operand.

(c) When the left operand is an **int**, we make them non-member operators, with two arguments, the left being an **int** and the right a **bigint**. Note that the implementation of these functions, **+**, **=**, *****, **/**, **%**, **==**, **!=**, **<**, **<=**, **>**, **>=**, are all identical, namely that they cast the **int** to a **bigint** and call the other operator.

(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. While it is a member of **bigint**, it behaves like a non-member, using only other functions.

(f) The relational operators all trivially call a **compare** function, which is private, and needs to be rewritten.

(g) The / and % functions call **divrem**, which is private. One can not produce a quotient without a remainder, and vice versa.

(h) The operator * and **divrem** use the Ancient Egyptian multiplication and division algorithms, which only need the ability to add and subtract, and use a stack. Presumably Pharaoh's engineers used this algorithm. Fill in the missing code.

(i) Finally, 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

```
int small_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 4622 would be stored in a vector v as: $v_3 = 4$, $v_2 = 6$, $v_1 = 2$, $v_0 = 2$. In other words, if a digit's value is $d \times 10^k$, then $v_k = d$.

(c) Then use **grep** or your editor's search function to find all of the occurrences of **small_value**. Each of these occurrences needs to be replaced.

(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) Change all of the constructors so that instead of initializing **small_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.

(g) Add two new private functions **do_bigadd** and **do_bigsub**.

(h) 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.

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

(j) 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.

(k) 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.

(l) To implement **compare**, return a value that is < 0, = 0, or > 0, to show the relationship. First check the signs. If different, you immediately know which inequality to return. If the same, and for positive numbers, the longer vector (with more digits) is greater than the shorter one. If they are the same length, start comparing digits from the (high-order) end of the vector to the (low-order) front for a difference, For negative numbers, the smaller number is greater. which tells you the inequality. Otherwise return equal. This assumes that vectors are stored in a canonical manner without high-order zeros.

(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:

```
% dc
99999 40^p
99960007799012091383420303819357209019506959637410544273315009185275\
28345614337549894898970052160072539374482497169939887012095083298830\
62070365779361889044985647838373419929138990120007799996000001
```

5. 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.

To check for memory leak, use **dbx** interactively, but you can also test your program with **bcheck -all ydc**. **bcheck**(1) is a shell script that runs **dbx**(1). Unfortunately on the Intel CPUs it is rather slow.

6. 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. Use **valgrind** to check for memory leaks.

If you are doing pair programming, follow the additional instructions in **/afs/cats.ucsc.edu/courses/cmps012b-wm/Syllabus/pair-programming/** and also submit **PARTNER**.