`$Id: asg1-dc-bigint.mm,v 1.265 2022-03-31 18:47:16-07 - - $`
`/afs/cats.ucsc.edu/courses/cse111-wm/Assignments/asg1-dc-bigint`
`https://www2.ucsc.edu/courses/cse111-wm/:/Assignments/asg1-dc-bigint/`

1. Using C++11/14/17 (`g++ -std=gnu++20`)

All programming in this course will be done C++ style, not C style.

Do not use :	Instead, use :
`char*` strings	**`<string>`**
C arrays	**`<vector>`**
`<stdio.h>`, **`<cstdio>`**	**`<iostream>`**, **`<iomanip>`**
pointers	**`<shared_ptr>`** or **`<unique_ptr>`**
`union`	inheritance or **`<variant>`**
<*header***.h>**	**<c***header***>**

Include only C++ header files and use the declaration **`using namespace std;`** Include **<c***header***>** files only when C++ header files do not provide a necessary facility. Include **<***header***.h>** files from C only when an appropriate **<c***header***>** file does not exist. Use the script **`cpplint.py.perl`** (a wrapper for **`cpplint.py`**) to check style.

The production system for all work is **`unix.ucsc.edu`** using **`g++`**. Compile with

```
g++ -std=gnu++20 -g -O0 -Wall -Wextra -Wpedantic -Wshadow -Wold-style-cast
```

Following is a description of these options :

- **`-std=gnu++20`** Gnu dialect of C++20.
- **`-g`** produces debugging information into object files and the binary executable. This is necessary for **`gdb`** and **`valgrind`** to use symbolic names.
- **`-O0`** reduces compilation time and makes debugging produce more expected results. Optimization may rearrange bugs in code in unexpected ways.
- **`-Wall`** enables all the warnngs about questionable constructions.
- **`-Wextra`** enables extra warnings that are not enabled with **`-Wall`**.
- **`-Wpedantic`** issues all warnings required by strict ISO C++ and rejects all programs that do not conform to ISO C++.
- **`-Wshadow`** warns whenever a local variable or declaration shadows another variable, parameter, or class member.
- **`-Wold-style-cast`** warns about the use of any old-style (C-style) cast. Instead, use one of : **`static_cast`**, **`dynamic_cast`**, **`const_cast`**, **`reinterpret_cast`**. Better yet, code in such a way as to not need casts.
- **`-fdiagnostics-color=never`** prevents the compiler from using those silly annoying colors in diagnostics.

The particular **`g++`** compiler we will be using is

```
-bash-1$ which g++
/opt/rh/devtoolset-11/root/usr/bin/g++
-bash-2$ g++ --version | grep -i g++
g++ (GCC) 11.2.1 20210728 (Red Hat 11.2.1-1)
-bash-3$ uname -npo
unix2.lt.ucsc.edu x86_64 GNU/Linux
```

If you develop on your personal system, be sure to port and test your code on the Linux timeshares. If it compiles and runs on your system, but not on the timeshares, ***then it does not work.***

2. Overview

This assignment will involve overloading basic integer operators to perform arbitrary precision integer arithmetic in the style of **dc**(1). 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
```

If you have any questions as to the exact format of your output, just run **dc**(1) and make sure that, for the operators specified above, your program produces exactly the same output. A useful program to compare output from your program with that of **dc**(1) is **diff**(1), which compares the contents of two files and prints only the differences. Also look in the subdirectory **misc/** for some examples.

See also:

- dc (computer program)
  https://en.wikipedia.org/wiki/Dc_(computer_program)
- dc, an arbitrary precision calculator
  https://www.gnu.org/software/bc/manual/dc-1.05/html_mono/dc.html

3. Implementation strategy

As before, you have been given starter code.

(a) **Makefile**, **debug**, and **util** 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 debug mode. This is strictly for debugging.

(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.

4. 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) Most of the functions take a arguments of type **`const bigint&`**, i.e., a constant reference, for the sake of efficiency. But they have to return the result by value.

(b) 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. **`operator<<`** is used by debugging functions.

(c) The function **`print`** (suitably modified) is used for actual output.

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

(e) All of the functions of **`bigint`** only work with the sign, using **`ubigint`** to do the actual computations. So **`bigint::operator+`** and **`bigint::operator-`** will check the signs of the two operands then call **`ubigint::operator+`** or **`ubigint::operator-`**, as appropriate, depending on the relative signs and magnitudes. The multiplication and division operators just call the corresponding **`ubigint`** operators, then adjust the resulting sign according to the rule of signs.

5. Class `ubigint`

Class **`ubigint`** implements unsigned large integers and is where the computational work takes place. Class **`bigint`** takes care of the sign. Now we turn to the representation of a **`ubigint`**, which will be represented by vector of bytes.

(a) Replace the declaration

```
using ubigvalue_t = unsigned long;
```

with

```
using ubigvalue_t = vector<uint8_t>;
```

in the header file **`<ubigint.h>`**. The type **`uint8_t`** is an unsigned 8-bit integer defined in **`<cstdint>`**.

(b) In storing the big integer, each digit is kept as an integer in the range 0 to 9 in an element of the vector. 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 **`v[k]==d`**, then the digit's place value is **`d*pow(10,k)`**. In mathematical notation, the value of a radix 10 (base 10) number v with n digits is:

$$\sum_{i=0}^{n-1} v_i 10^i = v_{n-1}10^{n-1} + v_{n-2}10^{n-2} + \cdots + v_2 10^2 + v_1 10^1 + v_0 10^0$$

(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() > 0 and back() == 0) pop_back();
```

(d) Canonical form.
- Zero is represented as a vector of size zero and a positive sign.
- High-order zeros are suppressed.
- All digits are stored as **`uint8_t`** values in the range 0...9, not as characters in the range `'0'`...`'9'`.
- To print a digit, cast it to an integer : **`cout << static_cast<int> (digit)`**.
- This can be done more easily by : **`cout << int (digit)`**, which looks like a ctor call.

(e) 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.

6. Implementation of operators

(a) For **`bigint::operator+`**, check the signs.

(1) If the signs are the same :
- Call **`ubigint::operator+`** with the unsigned numbers.
- The sign of the result is the sign of either number.

(2) If the signs are different :
- Call **`ubigint::operator-`** with the larger number as its left number.
- The sign of the result is the sign of the larger number.

(b) The operator **`bigint::operator-`**, check the signs.

(1) If the signs are different :
- Call **`ubigint::operator+`** with the unsigned numbers.
- The sign of the result is the sign of the left number.

(2) If the signs are the same :
- Call **`ubigint::operator-`** with the larger number as its left number.
- If the left number is larger, the sign of the result is its sign.
- Else the the result has the opposite of the sign of the right number.

(c) For the above **`bigint::operator+`** and **`bigint::operator-`**, to find the "larger" number, make use of **`ubigint::operator<`**. Since the numbers are kept in *canonical form* (see above), to compare them :

(1) Check the **`size()`** of each vector. If different, the larger number has the greater size.

(2) If the sizes are the same, write a loop iterating from the highest-order digit toward the lowest-order digit, comparing digit by digit.
- As soon as a difference is found, return true or false, as appriate.
- If all digits are the same, then return false.

(d) To implement **`ubigint::operator+`**, create a new **`ubigint`** and proceed from the low order end to the high order end, adding digits pairwise. For any sum >= 10, take the remainder and add the carry to the next digit. Use **`push_back`** to append the new digits to the **`ubigint`**. 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 `ubigint::operator-`, also create a new empty vector, starting from the low order end and continuing until the high end. If the left digit is smaller than the right digit, the subtraction will be less than zero. In that case, add 10 to the digit, and set the borrow to the next digit to −1. 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) 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.

(g) 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.

(h) Implement function `bigint::operator*`, which uses the rule of signs to determine the result. The number crunching is delegated to `ubigint::operator*`, which produces the unsigned result.

(i) Multiplication in `ubigint::operator*` proceeds by allocating a new vector whose size is equal to the sum of the sizes of the other two operands. If `u` is a vector of size `m` and `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 `p` as you would by hand. The algorithm can be described as follows:

```
p = all zeros
for i in interval [0,m):
   carry = 0
   for j in interval [0,n):
      digit = p[i+j] + u[i] * v[j] + carry
      p[i+j] = digit % 10
      carry = digit / 10
   p[i+n] = carry
```

Note that the interval `[a,b)` refers to the half-open interval including `a` but excluding `b`. This is the set `{x| a<=x && x<b}`. In the same way, a pair of iterators in C++ is used to bound an interval (begin and end pair).

(j) 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 `vector`s 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`

(k) 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.

(l) The unsigned division function that is provided depends on two private functions, **multiply_by_2** and **divide_by_2**, which are in-lace non-constant functions. They both perform **without** creating a new object.

   (1) To implement **multiply_by_2**, iterate from the low order digit, and double each digit (remainder 10), carrying to the next higher digit. At the end, if the carry is 1, use **push_back**.

   (2) To implement **divide_by_2**, iterate from the low order digit, and divide each digit by 2. Then, if the next higher digit is odd, add 5 to the current digit. Be careful of the end, and **pop_back** any remaining 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.

(n) The function **print** will print numbers in the same way as **dc**(1) does.

(o) The **pow** function 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. The power function is not a member of either **bigint** or **ubigint**, and is just considered a library function that is implemented using more primitive operations.

7. Memory leak and other problems

Make sure that you test your program completely so that it does not crash on a Segmentation Fault or any other unexpected error. Since you are not using pointers, and all values are inline, there should be no memory leak. Use **valgrind**(1) to check for and eliminate uninitialized variables and memory leak.

If your program is producing strange output or segmentation faults, use **gdb**(1) and the debug macros in the **debug** module of the **code/** subdirectory.

8. 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 to verify basic formatting.

Look in the grader's score subdirectory for instructions to graders. Read **Syllabus/pair-programming/** and also submit **PARTNER** if you are doing pair programming. Either way submit the **README** described therein.

9. Et cetera (καὶ τὰ ‘έτερα).

The accuracy of the Unix utility **dc**(1) can be checked by :

```
echo '82 43/25 43+65P80P82P73P76P32P70P79P79P76P10P' | dc
```